



.: [OllyDbg] :.



.- Análisis de sus posibilidades .-

• Un proyecto aún por hacer

Hace un tiempo, cuando apareció el OllyDbg y tuve la ocasión de probarlo, en seguida me saltó un pensamiento: no es más que una versión actualizada del win32dasm. Sin embargo, a medida que lo estaba utilizando, veía que, por lo pronto, tenía muchísimas más opciones que el win32dasm y que por ello en ocasiones se hacía incluso un poco deficiente e incómodo de usar.

También es cierto que pronto entendí que era algo más potente que el win32dasm... pero seguía siendo incómodo para aquellos que estábamos usando durante largos períodos de tiempo el SoftIce y/o el TRW2000 (auténticas maravillas ;o).

Pronto me hice con el control del OllyDbg, y comencé a estudiar sus posibilidades... que si un BreakPoint condicional con *logging*, que si modifico una sección de memoria, que si referencias a cadena, que si parcheo sobre la marcha..... y todo ello sin detener bruscamente el sistema, con la posibilidad de seguir conectado, y con una estabilidad que ya quisiera para sí haber soñado en alguna ocasión el win32dasm.

Pronto empecé por leer partes del manual (archivo .hlp)... y fue el primer encuentro impresionante! Un manual que más o menos cumple con su cometido, pero que ¿cuántos se lo han leído? Quizás por estar en inglés, quizás porque con cuatro comandos podemos hacer casi todo, quizás porque hay muchas opciones que no sabemos para qué son ni nos interesan... Sea como fuere, el principal proyecto que tuve en mente fue traducir todo el fichero de ayuda. Y ahí el primer problema ¿cómo se pueden traducir términos como “step into”-“step over”? Se perdería la esencia real del *debugging*. Así pues, lo que he decidido ha sido ir haciendo un análisis a todas las posibilidades que nos ofrece OllyDbg, y que nos pueden ayudar a hacer más sencilla la tarea de trazar algunos programas.

• Algunas consideraciones

A la hora de hacer este pequeño tutorial no quiero que sea enfocada sólo hacia las técnicas de cracking, porque tampoco un debugger se orienta únicamente hacia eso. Se trata de ver qué añade este debugger que nos pueda ayudar a la hora de depurar nuestros propios programas.

En todo momento usaré algunos términos que daré por conocidos, o al menos por entendidos, por lo que no me pararé a explicarlos. En ocasiones usaré traducciones al castellano, que posiblemente a quienes sean de América Latina les pueda resultar un poco extraño, pero que en el ámbito español son entendidas. Habrá también términos que usaré indistintamente (trazar y tracear, depurar y debuggear, ejecutar y correr, dumppear y volcar, etc...).

A la hora de describir una selección de menú (y submenús) lo indicaré separando las opciones con una barra vertical, por ejemplo: `File|Attach` significa que se ha seleccionado la opción `Attach` del menú `File`.

Si la opción nos lleva a un diálogo, las opciones del diálogo las indicaré separándolas con un guión (-), indicando que se trata de una pestaña mediante un slash (/). Por ejemplo: `Options|Debuggin options - Analysis1/Analyse code structure`, significaría: al pulsar la opción `Debuggin options` del menú `Options` sale un diálogo; en la pestaña `Analysis1`, hay un control llamado `Analyse code structure`. En todo momento estas indicaciones las marcaré con un tipo de letra de espaciado fijo.

• De qué está compuesto el OllyDbg y cómo funciona

Basicamente el OllyDbg es un debugger para aplicaciones de win32, y como tal tiene prácticamente todo lo necesario para poder depurar éstas. Hay que tener en cuenta que no es un debugger que actúe en ring0, sino que lo hace en ring3, por lo que no tiene un control total del sistema como pueden tener otros debugger (SoftIce o TRW2000), aunque también hay que reconocer que el modo en que lo hace en ring3 es estupendo.

Debido a ello el debugger corre simultaneamente con el programa, permitiéndonos hacer una de las cosas que con otros debuggers es prácticamente imposible: mientras el programa a depurar está ejecutándose, modificar datos de la memoria del propio proceso del programa.

Como debugger que es podremos examinar desde el OllyDbg: qué módulos se están ejecutando, y qué módulos externos están linkados (DLL) al módulo principal, hacer búsquedas y modificar bloques de memoria pertenecientes al propio proceso así como en los heaps generador por el propio proceso, si es un proceso multihilo también podremos depurar cada hilo por separado; ver también las ventanas y controles creados (con su handle, jerarquías, procesos, etc...), una pila interna de las llamadas a funciones (*call stack*), así como una ventana de referencias a comandos, constantes, punteros, cadenas y un visor de expresiones. Finalmente también podremos ver el modulo en formato raw (como está en el fichero antes de ser mapeado en memoria), la ventana para configurar los puntos de ruptura (*breakpoints*) y el estado de la CPU, que incluye varias partes: desensamblado, registros, pila y volcado de memoria.

Hasta aquí podría parecer un debugger más de tantos que hay, pero a esto tenemos que añadirle que el OllyDbg no sólo está orientado a ejecutar paso a paso un programa, sino que está pensado para analizar bloques de código, para lo que añade un analizador de código que nos va a facilitar la labor considerablemente, pudiendo editar/añadir etiquetas, comentarios, mostrando pequeñas ayudas en tiempo real, y un sinfín de trucos y ayudas que harán de nuestras sesiones de depuración algo agradable.

Cuando abandonamos la depuración de un módulo, el OllyDbg guarda en un archivo `.UDD` toda la información que considera importante (nombres simbólicos, comentarios, puntos de ruptura, etc...), que cargará la próxima vez que depuremos ese módulo. Esto es un fichero EXE quizás no tenga gran importancia, pero sí que lo tiene en un DLL, especialmente si es usado por varios EXEs, pues esa información se cargará cadavez que depuremos ese DLL (independientemente del EXE que haya linkado el DLL).

• El analizador de código

Cuando abrimos un módulo (normalmente un archivo .EXE) para depurarlo, el OllyDbg hace un análisis de las secciones que habitualmente son de código ejecutable. Para ello el programa dispone de una serie de opciones que podemos configurar, para que el análisis se ajuste lo más posible a nuestras necesidades¹. Lo que hace el analizador de código es “formatear” y añadir comentarios automáticos a las líneas de código, para que así nos sea más legible y sencillo el estudio y la comprensión.

El Olly en su primera pasada de análisis, es capaz de reconocer varias partes del código. En primer lugar reconoce puntos de entrada de subrutinas, para lo cual lo que hace es mirar qué direcciones son llamadas desde tres puntos (o más) del programa. De este modo, trazando los jumps y los calls puede reconocer casi con total seguridad todos los comandos. Añade además 20 métodos heurísticos.

Para la detección de funciones (procedures), tenemos la opción de indicar qué tipo de procedimiento queremos que use: Strict, heuristical o fuzzy². Una función (*procedure*) para el analizador es cualquier parte contigua de código que teniendo un punto de entrada puede pasar (al menos teóricamente) por todas las instrucciones siguientes.

Strict: entiende que la función tiene un punto de entrada, y al menos un punto de salida (return).

Heuristical: Este modo asume el punto de entrada por sí mismo.

Fuzzy: cualquier trozo más o menos consistente de código es considerado como una función, y estudiado y analizado como tal.

```

00470C03 |$ 833D F01A4F01 cmp dword ptr ds:[4F1AF0],1
00470C0A | 75 05          jnz short OTSJDJ.00470C11
00470C0C | E8 425C0000   call OTSJDJ.00476853
00470C11 |> FF7424 04     push dword ptr ss:[esp+4]
00470C15 | E8 725C0000   call OTSJDJ.0047688C
00470C1A | 68 FF000000   push 0FF
00470C1F | FF15 F8C24901 call dword ptr ds:[49C2F8]
00470C25 | 59           pop ecx
00470C26 | 59           pop ecx
00470C27 | C3          retn
00470C28 |$ 833D F01A4F01 cmp dword ptr ds:[4F1AF0],1
00470C2F | 75 05          jnz short OTSJDJ.00470C36
00470C31 | E8 1D5C0000   call OTSJDJ.00476853
00470C36 |> FF7424 04     push dword ptr ss:[esp+4]
00470C3A | E8 4D5C0000   call OTSJDJ.0047688C
00470C3F | 59           pop ecx
00470C40 | 68 FF000000   push 0FF
00470C45 | FF15 F4C14701 call dword ptr ds:[<&KERNEL32.ExitProce
00470C4B | C3          retn
00470C4C |$ 56           push esi
00470C4D | 6A 0C         push 0C
00470C4F | FR 0A17A000   call OTS.INI.00472424

```

OllyDbg es capaz de indicarnos el punto de entrada y las funciones por bloques (*procedures*)

De cualquier modo, las posibilidades de pérdida de información, y de no detectar algunas funciones es bastante alta, si bien algunos compiladores dejar todo bastante bien estructurado y entendible por el OllyDbg.

¹ He de advertir que estoy trabajando con la versión 1.08b, y que esta parte del OllyDbg está en continuo desarrollo y mejora; quizás lo que exponga en esta parte quede obsoleto o mejorado en versiones sucesivas.

² Para seleccionar qué tipo de análisis queremos podemos hacerlo en: Options|Debuggin Options - Analysis1.

OllyDbg también analiza fragmentos de código contiguo y recurrente, es decir, bucles (loops). Estos bloques corresponderían a sentencias de alto nivel de tipo “do” y “while” (en algunos compiladores también a “for”). OllyDbg no solo reconoce los bucles sencillos, sino también bucles anidados (unos dentro de otros). Los bucles son marcados con un corchete a la izquierda del código, indicando con un pequeño triángulo el punto de entrada, en el caso en que no coincidiese con el inicio propio del bucle.

```

00401910 | * 3302 | xor edx,edx
00401912 | . 8BFA | mov edi,edx
00401914 | . 42 | inc edx
00401915 | . C705 C142400 | mov dword ptr ds:[4042C1],0
0040191F | > E9 33010000 | jmp ACN.00401A57
00401924 | > 3B15 C942400 | cmp edx,dword ptr ds:[4042C9]
0040192A | > 0F84 26010000 | je ACN.00401A56
00401930 | . 33DB | xor ebx,ebx
00401932 | . 43 | inc ebx
00401933 | > E9 12010000 | jmp ACN.00401A4A
00401938 | > 3B1D C942400 | cmp ebx,dword ptr ds:[4042C9]
0040193E | > 0F84 FB000000 | je ACN.00401A3F
00401944 | . 3BDA | cmp ebx,edx
00401946 | > 0F84 F3000000 | je ACN.00401A3F
0040194C | . FF35 C942400 | push dword ptr ds:[4042C9]
00401952 | . 6A 01 | push 1
00401954 | . E8 290C0000 | call ACN.00402582
00401959 | . 0FB608 | movzx ecx,byte ptr ds:[eax]
0040195C | . 53 | push ebx
0040195D | . 6A 01 | push 1
0040195F | . E8 1E0C0000 | call ACN.00402582
00401964 | . 0FB608 | movzx eax,byte ptr ds:[eax]

```

Dentro de un proceso, puede encontrar bucles, y dentro de estos, otros bucles anidados.

```

0040198E | > EB 59 | jmp short ACN.004019E9
00401990 | > FF35 C942400 | push dword ptr ds:[4042C9]
00401996 | . FF35 C542400 | push dword ptr ds:[4042C5]
0040199C | . E8 E10B0000 | call ACN.00402582
004019A1 | . 0FB608 | movzx ecx,byte ptr ds:[eax]
004019A4 | . 53 | push ebx
004019A5 | . FF35 C542400 | push dword ptr ds:[4042C5]
004019AB | . E8 D20B0000 | call ACN.00402582
004019B0 | . 0FB608 | movzx eax,byte ptr ds:[eax]
004019B3 | . 52 | push edx
004019B4 | . F7E1 | mul ecx
004019B6 | . 8BC8 | mov ecx,eax
004019B8 | . 5A | pop edx
004019B9 | . 52 | push edx
004019BA | . FF35 C542400 | push dword ptr ds:[4042C5]
004019C0 | . E8 BD0B0000 | call ACN.00402582
004019C5 | . 0FB608 | movzx eax,byte ptr ds:[eax]
004019C8 | . 03C1 | add eax,ecx
004019CA | . 3B05 D142400 | cmp eax,dword ptr ds:[4042D1]
004019D0 | > 74 0C | je short ACN.004019DE
004019D2 | > C705 C142400 | mov dword ptr ds:[4042C1],-1
004019DC | > EB 13 | jmp short ACN.004019F1
004019DE | > FF05 C542400 | inc dword ptr ds:[4042C5]
004019E4 | > A1 C542400 | mov eax,dword ptr ds:[4042C5]
004019E9 | > 3A05 2940400 | cmp al,byte ptr ds:[404029]
004019EF | > 76 9F | jbe short ACN.00401990
004019F1 | > 833D C142400 | cmp dword ptr ds:[4042C1],0
004019F8 | > 75 45 | jnz short ACN.00401A3F
004019FA | . BF 01000000 | mov edi,1

```

Si el bucle no se inicia en la primera instrucción de éste, es indicado el punto de entrada con un pequeño triángulo.

De igual modo el analizador es capaz de interpretar secuencias compiladas de tipo “switch” (*case*). En ocasiones estas instrucciones son compiladas con restas de diferentes constantes, lo que hace que seguirlas mentalmente sea casi una tarea imposible. OllyDbg intenta hacer un trazado (con bastante éxito) en estas tablas, haciendo referencias y a cada una de ellas.

0047A696	. 5E	pop esi	KERNEL32.BFF8B560
0047A697	> 8A1F	mov bl,byte ptr ds:[edi]	
0047A699	. 47	inc edi	
0047A69A	. 83F8 0B	cmp eax,0B	Switch (cases 0..B)
0047A69D	√ 0F87 77020000	ja OTSJDJ.0047A91A	
0047A6A3	> FF2485 E1AA4	jmp dword ptr ds:[eax*4+47AAE1]	Case 0 of switch 0047A69A
0047A6AA	> 80FB 31	cmp bl,31	
0047A6AD	√ 7C 0C	j1 short OTSJDJ.0047A6BB	
0047A6AF	. 80FB 39	cmp bl,39	
0047A6B2	√ 7F 07	jg short OTSJDJ.0047A6BB	
0047A6B4	> 6A 03	push 3	
0047A6B6	√ E9 1D020000	jmp OTSJDJ.0047A8D8	
0047A6BB	> 3A1D 14C94900	cmp bl,byte ptr ds:[49C914]	
0047A6C1	√ 75 07	jnz short OTSJDJ.0047A6CA	
0047A6C3	> 6A 05	push 5	
0047A6C5	√ E9 46020000	jmp OTSJDJ.0047A910	
0047A6CA	> 0FBEC3	movsx eax,bl	
0047A6CD	. 83E8 2B	sub eax,2B	Switch (cases 2B..30)
0047A6D0	√ 74 1E	je short OTSJDJ.0047A6F0	
0047A6D2	. 48	dec eax	OTSJDJ.<ModuleEntryPoint>
0047A6D3	. 48	dec eax	OTSJDJ.<ModuleEntryPoint>
0047A6D4	√ 74 0E	je short OTSJDJ.0047A6E4	
0047A6D6	. 83E8 03	sub eax,3	
0047A6D9	√ 0F85 D4020000	jnz OTSJDJ.0047A9B3	
0047A6DF	> E9 8F000000	jmp OTSJDJ.0047A773	Case 30 ('0') of switch 004
0047A6E4	> 6A 02	push 2	Case 2D ('-') of switch 004
0047A6E6	. C745 D8 0080	mov [local.10],8000	
0047A6ED	. 58	pop eax	KERNEL32.BFF8B560
0047A6EE	^ EB A7	jmp short OTSJDJ.0047A697	
0047A6F0	> 8365 D8 00	and [local.10],0	Case 2B ('+') of switch 004
0047A6F4	. 6A 02	push 2	
0047A6F6	. 58	pop eax	KERNEL32.BFF8B560
0047A6F7	. 58	pop esi	

Las referencias de tipo “case” siempre nos indican a qué switch pertenecen

En ocasiones intentará sugerir el significado de cada caso, facilitándonos la localización y la intención de la tabla. En la última figura se ve cómo el propio OllyDbg sugiere 30 = '0', y 2D = '-'. En otros casos puede mostrarnos nombres simbólicos de constantes del sistema: WM_PAINT, EXCEPTION_ACCESS_VIOLATION, etc.

En ocasiones una sucesión de sentencias de tipo “if” queda compilada como una serie de comparaciones sin modificación de los registros, por lo que realmente no es un switch, pero podría ser evaluado como tal. Podemos indicar a OllyDbg que este tipo de “if” encadenados los interprete como un switch³.

OllyDbg contiene además la descripción interna de más de 1900 API's de uso común, así como los nombres simbólicos de sus constantes. En la lista se han incluido las API's de: kernel32, gdi32, user32, advapi32, comdlg32, shell32, version, shlwapi, comctl32, winsock, ws2_32 y msvcrt. Además, se pueden añadir nuevas descripciones de API's (el modo de hacer esto lo veremos más adelante).

Cuando el analizador encuentra una llamada (mediante un `call` directo, o mediante un `call` referenciando un `jmp`), intenta decodificar los datos previos que se empujan a la pila mediante `push`, intentando así decodificar las constantes que se introducen como argumentos para la API. OllyDbg tiene además las descripciones de otras 400 funciones comunes del lenguaje C.

Cuando la opción “Guess number of arguments of unknown functions” está activada el analizador intenta descubrir cuántos `dwords` se empujan a la pila como argumentos para una función, y los interpreta como Arg1, Arg2, ... reflejándolo así en el código. Normalmente OllyDbg traza el contenido y el valor de los registros, siempre que sea un fragmento de código lineal, quedando rota la “linealidad” con sentencias de tipo `jmp` hacia fuera del bloque analizado.

³ Podemos activar esta opción en Options|Debugging Options - Analysis1/Decode cascaded IF's as switch

De igual modo el analizador puede reconocer datos entre funciones, cadenas de texto, o simplemente pequeños trucos para ofuscar el código real del programa.

En ocasiones podemos encontrarnos que tenemos los archivos objeto (obj, de formato OMF y COFF) del ejecutable, en el cual está contenida la información acerca de nombres simbólicos de constantes, variables, nombres de funciones, etc. En esos casos, si pulsamos con el botón derecho del ratón sobre la parte del código, en el apartado *Analysis* podemos seleccionar *Scan Object files*, que nos permite seleccionar los archivos .obj, pudiendo crear hasta 4 grupos, para que el analizador busque coincidencias entre el código del módulo y la información del fichero objeto.

De forma análoga en ocasiones las librerías DLL no crean la tabla de exportación usando el nombre de las APIs, sino mediante un número (*ordinals*). Si tenemos acceso a las librerías de importación de esa DLL (*implib*), se lo podremos indicar a OllyDbg para que cargue la información de la librería de importación, substituyendo los números de las APIs por su nombre simbólico⁴. Para ello solo tenemos que seleccionar: `Debug|Select import libraries`.

• Opciones de configuración del analizador de código

En el apartado `Options|Debugging options`, tenemos dos pestañas llamadas *Analysis1* y *Analysis2* para indicar diferentes opciones para que se comporte de un modo u otro el analizador. La primera de las pestañas está orientada a la interpretación código de cara al usuario, para que el assembler sea lo más comprensible posible, y es donde le podremos indicar todas las opciones que hemos visto hasta este punto.

La segunda pestaña está orientada a indicar qué instrucciones ha de considerar inválidas OllyDbg, para facilitarle la tarea de encontrar partes de datos dentro del código, y que no han de ser mostradas como instrucciones, sino como datos en crudo. Hay muchas instrucciones que aunque son válidas para el procesador apenas aparecen en un programa válido para ser ejecutado en win32. Normalmente el analizador comenta estas instrucciones como posiblemente inválidas, y son tratadas por ello como datos binarios, y no como instrucciones. En este panel se pueden seleccionar instrucciones inválidas que queremos aceptar como válidas.

```

00470CE2 |< EB 02      jmp short OTSJDJ.00470CE6
00470CE4 |> 33C0      xor eax,eax
00470CE6 |> 5F       pop edi
00470CE7 |. 5E       pop esi
00470CE8 |. 5B       pop ebx
00470CE9 |. C3       retn
00470CEA |. CC       int3
00470CEB |. CC       int3
00470CEC |. CC       int3
00470CED |. CC       int3
00470CEE |. CC       int3
00470CEF |. CC       int3
00470CF0 |< 51       push ecx
00470CF1 |. 3D 00100000  cmp eax,1000
00470CF6 |. 8D4C24 08  lea ecx,dword ptr ss:[esp+8]
00470CFA |> 72 14     jb short OTSJDJ.00470D10
00470CFC |> 81E9 00100000  sub ecx,1000
00470D02 |. 2D 00100000  sub eax,1000
00470D07 |. 8501     test dword ptr ds:[ecx],eax
00470D09 |. 3D 00100000  cmp eax,1000
00470D0E |> 73 EC     jnb short OTSJDJ.00470CFC
00470D10 |> 2BC8     sub ecx,eax

```

OllyDbg detecta grupos de bytes que realmente son alineaciones de código entre funciones.

⁴ Un ejemplo de este tipo de librerías es MFC42.DLL.

Veamos que hacen cada una de las opciones de la pestaña de `Analysis1`.

Procedure recognition: esta opción se ha explicado ya anteriormente.

Show ARGs and LOCALs in procedures: En las funciones es habitual que los compiladores al entrar en ellas reserve un espacio temporal en la pila, usando los registros ESP y EBP, preservando el estado previo de la pila, para que al finalizar la función pueda liberar ese espacio temporal.

Es habitual que en el código de la función se referencien las variables locales mediante [EBP-offset], y a los argumentos de la función mediante [EBP+offset]; es decir, EBP marcaría el ecuador entre la pila al entrar a la función y el espacio reservado.

Si tenemos esta opción activada OllyDbg substituye las referencias de este tipo por nombres simbólicos de tipo [ARG.1], [ARG.2], y [LOCAL.1], [LOCAL.2], etc... que nos evita tener que memorizar los offsets de cada variable, o de cada argumento.

Show arguments of known functions: Muestra los valores de los argumentos, así como el tipo de argumento que es, de las funciones que el propio OllyDbg tiene como definidas, y a las que antes hemos hecho mención. Habitualmente son funciones del API de Windows, pero también reconoce unas 400 funciones del lenguaje C.

Guess number of arguments of unknown functions: Esta opción la he comentado antes. Básicamente es similar a la anterior, pero con la peculiaridad de que lo hace en las funciones de las que no tiene la información almacenada. Mediante el número de dwords reservados en memoria por la función calcula cuántos argumentos puede tener. De este modo nos muestra Arg.1, Arg.2, etc. si encuentra elementos que son empujados a la pila antes de los `call` a estas funciones.

Auto start analysis of main module: Sencillo... cuando esta opción está activada, al cargar el módulo principal en el OllyDbg iniciará el análisis del código automáticamente. Cuando esta opción está desactivada, podemos forzar que OllyDbg analice el código pulsando Ctrl+A, o seleccionando en el menú contextual en la zona de desensamblado: `Analysis|Analyse code`.

Analyse code structure: Esta opción activa en el analizador el reconocimiento de estructuras de tipo switch, así como los bucles.

Decode cascaded IFs as switches: De esta opción ya hemos hablado anteriormente. Si el analizador encuentra sentencias de tipo “if” encadenadas, las analiza como si fuesen sentencias switch.

Trace contents of registers: El analizador traza el contenido de los registros dentro de fragmentos de código que sean lineales. Los argumentos que se han obtenido trazando código se muestran con “=>”. Un ejemplo muy ilustrativo está contenido en el fichero de ayuda del propio OllyDbg.

```

004116E1 | . C8 000000    ENTER 0,0
004116E5 | . BB 22154500  MOV EBX,<JMP.&KERNEL32.GetPrivateProfile>
004116EA | . BE 1B174100  MOV ESI,OT.0041171B          ; ASCII "inifile.ini"
004116EF | . BF 27174100  MOV EDI,OT.00411727          ; ASCII "Key1"
004116F4 | . B9 31174100  MOV ECX,OT.00411731          ; ASCII "SectionName"
004116F9 | . 33C0        XOR EAX,EAX
004116FB | . 51          PUSH ECX
004116FC | . 56          PUSH ESI                      ; [ IniFileName => "inifile.ini"

```

```

004116FD | . 50          PUSH EAX          ; | Default => 0
004116FE | . 57          PUSH EDI          ; | Key => "Key1"
004116FF | . 51          PUSH ECX          ; | Section => "SectionName"
00411700 | . FFD3       CALL EBX          ; | GetPrivateProfileIntA
00411702 | . 8B4D 08     MOV ECX,[ARG.1]
00411705 | . 8901       MOV [DWORD DS:ECX],EAX
00411707 | . 59          POP ECX
00411708 | . 83C7 05     ADD EDI,5
0041170B | . 83C8 FF     OR EAX,FFFFFFFF
0041170E | . 56          PUSH ESI          ; | IniFileName => "inifile.ini"
0041170F | . 50          PUSH EAX          ; | Default => FFFFFFFF (-1.)
00411710 | . 57          PUSH EDI          ; | Key => "Key2"
00411711 | . 51          PUSH ECX          ; | Section => "SectionName"
00411712 | . FFD3       CALL EBX          ; | GetPrivateProfileIntA
00411714 | . 8B4D 0C     MOV ECX,[ARG.2]
00411717 | . 8901       MOV [DWORD DS:ECX],EAX
00411719 | . C9          LEAVE
0041171A | . C3          RETN

0041171B . 69 6E 69 66 69>ASCII "inifile.ini",0
00411727 . 4B 65 79 31 00>ASCII "Key1",0
0041172C . 4B 65 79 32 00>ASCII "Key2",0
00411731 . 53 65 63 74 69>ASCII "SectionName",0

```

Decode tricky code sequences: De momento OllyDbg es capaz de decodificar un pequeño truco que permite mezclar atributos de texto y argumentos de una función junto con el propio código, para no referenciar los argumentos mediante offsets. Es capaz así de reconocer instrucciones `call` seguidas inmediatamente por cadenas ASCII, así como instrucciones `push` combinadas con `jmp`'s. En el fichero de ayuda del OllyDbg viene un pequeño fragmento de código con sistaxis para el MASM que puede orientar acerca del funcionamiento de este ayuda.

Keep analysis between sessions: Cuando está activado, al terminar el depurado de un módulo, OllyDbg guarda un fichero con los datos producidos, cargándolo automáticamente la próxima vez que se analice el mismo módulo, evitando así tener que esperar el tiempo necesario para hacer de nuevo el análisis.

Veamos que hacen cada una de las opciones de la pestaña de `Analysis2`.

Far calls and returns: Las *far calls* (llamadas entre segmentos, entre privilegios or entre procesos) apenas son usadas en los programas de win32.

Modifications of segment registers: Los programas de win32 no suelen modificar los registros de segmentos.

Privileged commands: Son instrucciones que no se pueden usar en código de usuario (ring3), pero que sí pueden ser usadas en modo ring0.

I/O commands: Aunque Windows NT no permite a las aplicaciones comunes acceder a los puertos de entrada-salida directamente algunos drivers modifican el mapa de bits de permisos para entrada-salida, permitiendo el acceso. Win9x no es tan estricto en este campo, conviene activar esta opción si sospechamos que el programa que estamos depurando accede a los puertos de entrada-salida directamente.

Commands that are equivalent to NOP: Algunas instrucciones no hacen absolutamente nada, salvo consumir ciclos de reloj. Normalmente los compiladores evitan el uso de estas instrucciones,

aunque puede haberlas si el fragmento de código está programado en ASM, y se han puesto para despistar. Cuando está activada esta opción el analizador entenderá estas instrucciones como tales, y no como datos.

Shifts out of range 1..31: Antes de hacer un shift la CPU realiza una máscara al contador con 1Fh, limitando así el rango del contador a 0..31. No obstante un shift con un contador de 0 no realiza ninguna operación, siendo casi imposible que sea generado por ningún coimpilador.

Superfluous prefixes: Marca esta opción para que el analizador entienda como instrucciones válidas aquellas que contienen prefijos repetidos o contradictorios, o aquellas que poseen prefijo de acceso a memoria pero sus argumentos son registros (por ejemplo `ES:XOR EAX,EAX`)

LOCK prefixes: Las aplicaciones normalmente no utilizan este prefijo, pues se utilizan funciones de la API del sistema que realizan la misma tarea, como `EnterCriticalSection`.

Unaligned stack operations: Los programas suelen manejar la pila con un alineamiento de dwords, que en win32 es el argumento básico por defecto para las funciones. Por ello es difícil encontrar en una aplicación instrucciones que manejen el puntero de la pila haciéndole modificar este alineamiento (`inc esp ; add esp,7`). Si marcamos esta opción el analizador interpretará estas instrucciones como tales, y no las analizará como datos binarios.

VxD calls: Las aplicaciones basadas en win9x usan las llamadas a VxD para comunicarse con los dispositivos virtuales. Estas llamadas son básicamente una `int 20` seguida por un dword que contiene el código del servicio requerido para el dispositivo. Las aplicaciones no llaman directamente a las VxD, pero si se intenta desensamblar un módulo de un driver podríamos toparnos con un galimatías de código. Esta opción está activada por defecto en los sistemas win9x.

• Un analizador de código de carne, hueso y materia gris

Bien, hasta aquí hemos visto qué puede hacer el OllyDbg con su analizador de código para facilitarnos la tarea del estudio y análisis de un bloque de código. En el próximo documento analizaré el desensamblador del OllyDbg, con toda la ventana de la CPU, que es quizás la más importante y la que tendremos casi constantemente activada mientras depuramos un programa.

De todos modos, y sin que sirva de adelanto, pienso que el análisis más importante es el que podamos hacer nosotros. Para ello OllyDbg ha dispuesto que podamos añadir nuestros propios apuntes sobre el código que él ha desensamblado y analizado.

OllyDbg nos permite añadir nombres simbólicos a direcciones de memoria concretas, así como comentarios al código. Si pulsamos sobre una línea de código con el botón derecho del ratón (menú contextual), podremos seleccionar `Label` y `Comment` (los atajos de teclado son “:” y “;” respectivamente). Mediante `Label` podremos asignar un nombre a la dirección concreta, viendo después en el desensamblado el nombre que le hayamos asignado en lugar del número de la dirección. De igual modo podemos añadir un comentario, que aparecerá en la parte derecha del código.

00401000	BA FF000000	mov edx,0FF	
00401005	8915 00204000	mov dword ptr ds:[402000],edx	
00401008	B8 00204000	mov eax,MAIN.00402000	
00401010	8A18	mov bl,byte ptr ds:[eax]	
00401012	FEC3	inc bl	
00401014	8818	mov byte ptr ds:[eax],bl	
00401016	80FB 20	cmp bl,20	
00401019	75 00	jnz short MAIN.00401028	
0040101B	8028 20	sub byte ptr ds:[eax],20	si es 20h pone el counter a 0
0040101E	40	inc eax	MAIN.<ModuleEntryPoint>
0040101F	3D 04204000	cmp eax,MAIN.00402004	
00401024	75 EA	jnz short <MAIN.inicio del loop>	
00401026	EB 75	jmp short MAIN.0040109D	
00401028	A1 00204000	mov eax,dword ptr ds:[402000]	
0040102D	B9 03000000	mov ecx,3	
00401032	33D2	xor edx,edx	
00401034	99	cdd	
00401035	F7F9	idiv ecx	edx= ECX mod 3
00401037	85D2	test edx,edx	
00401039	74 07	je short MAIN.00401042	
0040103B	B8 00204000	mov eax,MAIN.00402000	
00401040	EB CE	jmp short <MAIN.inicio del loop>	
00401042	A1 00204000	mov eax,dword ptr ds:[402000]	
00401047	B9 0B000000	mov ecx,0B	
0040104C	33D2	xor edx,edx	
0040104E	99	cdd	
0040104F	F7F9	idiv ecx	edx=ECX mod 11
00401051	85D2	test edx,edx	
00401053	74 07	je short MAIN.0040105C	
00401055	B8 00204000	mov eax,MAIN.00402000	
0040105A	EB B4	jmp short <MAIN.inicio del loop>	
0040105C	B8 00204000	mov eax,MAIN.00402000	

Fragmento de código al que se le ha añadido una etiqueta “inicio del loop” en 401010; y varios comentarios.

00401000	BA FF000000	mov edx,0FF	
00401005	8915 00204000	mov dword ptr ds:[402000],edx	
00401008	B8 00204000	mov eax,MAIN.00402000	
00401010	8A18	mov bl,byte ptr ds:[eax]	
00401012	FEC3	inc bl	
00401014	8818	mov byte ptr ds:[eax],bl	
00401016	80FB 20	cmp bl,20	
00401019	75 00	jnz short MAIN.00401028	
0040101B	8028 20	sub byte ptr ds:[eax],20	
0040101E	40	inc eax	MAIN.<ModuleEntryPoint>
0040101F	3D 04204000	cmp eax,MAIN.00402004	
00401024	75 EA	jnz short MAIN.00401010	
00401026	EB 75	jmp short MAIN.0040109D	
00401028	A1 00204000	mov eax,dword ptr ds:[402000]	
0040102D	B9 03000000	mov ecx,3	
00401032	33D2	xor edx,edx	
00401034	99	cdd	
00401035	F7F9	idiv ecx	
00401037	85D2	test edx,edx	
00401039	74 07	je short MAIN.00401042	
0040103B	B8 00204000	mov eax,MAIN.00402000	
00401040	EB CE	jmp short MAIN.00401010	
00401042	A1 00204000	mov eax,dword ptr ds:[402000]	
00401047	B9 0B000000	mov ecx,0B	
0040104C	33D2	xor edx,edx	
0040104E	99	cdd	
0040104F	F7F9	idiv ecx	
00401051	85D2	test edx,edx	
00401053	74 07	je short MAIN.0040105C	
00401055	B8 00204000	mov eax,MAIN.00402000	
0040105A	EB B4	jmp short MAIN.00401010	
0040105C	B8 00204000	mov eax,MAIN.00402000	

El mismo fragmento de código, sin la etiqueta y sin los comentarios.