Herramientas

José María Alonso Cebrián
Jordi Gay Sensat
Antonio Guzmán Sacristán
Pedro Laguna Durán
Alejandro Martín Bailón
Jordi Serra Ruiz

PID_00208396

UOC
Universitat Oberta de Catalunya

Índice

Introducción..... 5

1. *Debuggers*..... 7
1.1. GDB 7
1.1.1. Primer contacto 8
1.1.2. Comandos básicos 8
1.1.3. Ejecución de un programa simple 19
1.1.4. Ejecutable con tabla de símbolos 19
1.1.5. Ejecutable sin tabla de símbolos 22
1.1.6. *Insight*..... 27
1.1.7. Conclusiones 32
1.2. OllyDbg 32
1.2.1. Primer contacto 33
1.2.2. Opciones 36
1.2.3. Modificar un programa en ejecución 39
1.2.4. Conclusiones 43

2. Compiladores/lenguajes..... 44
2.1. Generar "*opcodes*" 45

Introducción

Para poder analizar vulnerabilidades o *exploits* se necesitan conocimientos básicos de varias herramientas. En primer lugar se encuentran los *debuggers*. Estas aplicaciones permiten analizar, a bajo nivel (código máquina y ensamblador), la ejecución de las aplicaciones. Así es posible detectar vulnerabilidades y planear la escritura de programas que alteren la ejecución normal de otros.

En segundo lugar se encuentran los lenguajes de programación. Muchos de los *exploits* que existen están escritos en el lenguaje de programación C. Por otro lado, es muy necesario conocer conceptos básicos del lenguaje ensamblador para poder analizar aplicaciones a muy bajo nivel, cosa que proporciona un gran control sobre la ejecución de los programas.

1. *Debuggers*

Existen muchos *debuggers* en el mercado. Los hay que están muy extendidos, y otros que son populares. Bajo Linux el *debugger* más potente es el GDB[1] (*The GNU Project Debugger*). También se puede ejecutar en los sistemas UNIX más populares y en Microsoft Windows.

(1)**GDB**. The GNU Project Debugger. Disponible en: http://www.gnu.org/software/gdb/

En plataformas Windows existe un abanico de opciones en cuanto a *debuggers*, aunque para introducirse en el mundo del *debugging* en ensamblador, tal vez el más recomendado sea el OllyDbg[2]. Una vez adquiridos los conocimientos básicos con OllyDbg, pueden utilizarse fácilmente con otras herramientas como WinDbg[2].

(2)**OllyDbg**. Dispnible en: http://www.ollydbg.de/

En este módulo se verá cómo usar estas herramientas para controlar la ejecución de procesos a bajo nivel (código máquina y lenguaje ensamblador). En los ejemplos que se muestran no se ejecutan *exploits*, sino que se eligen pequeñas aplicaciones para ser modificadas en tiempo de ejecución directamente desde el *debugger*. Muchos *exploits* modifican las aplicaciones que atacan en tiempo de ejecución para conseguir ejecutar su propio código. Los ejemplos no son *exploits* en sí mismos, pero introducen el concepto de cambiar el comportamiento de las aplicaciones durante la ejecución de estas. El concepto es usado en muchos *exploits* que, utilizando vulnerabilidades de las aplicaciones, cambian su comportamiento para que hagan lo que el atacante desea.

1.1. GDB

Este es el *debugger* usado en plataformas GNU/Linux. Tiene una interfaz de comandos muy potente que permite realizar un conjunto muy extenso de operaciones. El modo de trabajo normal de gdb es bajo el intérprete de comandos. Existen varias aplicaciones que, comunicándose directamente con gdb, añaden una interfaz gráfica al *debugger* para hacerlo más manejable. Aunque estas interfaces son muy prácticas, la potencia de gdb reside en el intérprete de comandos.

Se presentan aquí las operaciones básicas para poder trazar la ejecución de aplicaciones con esta herramienta. Muchos lenguajes de alto nivel son soportados por esta herramienta, permitiendo el seguimiento a alto nivel de las aplicaciones si el código fuente está disponible. Aun así, tanto si el código fuente está disponible como si no, gdb puede hacer el seguimiento de la aplicación a bajo nivel (ensamblador).

1.1.1. Primer contacto

El entorno del *debugger* se puede invocar con:

```
$ gdb
```

De esta manera, se entra en el modo interactivo de comandos sin ningún programa cargado. A la llamada del *debugger* se pueden especificar comandos.

```
$ gdb programa
```

De esta manera, se carga el programa especificado, preparando todo el entorno para su ejecución. La ejecución del programa en sí no empezará hasta que se especifique apropiadamente en la línea de comandos. Hay dos maneras más de llamar al *debugger*. La primera de ellas es:

```
$ gdb programa core
```

Esta es la forma de llamar a gdb para trazar un estado de excepción de un programa que ha acabado normalmente. El fichero *core* contiene el estado de finalización del programa. De esta manera, se puede inspeccionar, a bajo nivel, qué es lo que causó la finalización inesperada del programa. Para generar los ficheros *core* el sistema tiene que estar configurado para ello. Este es un sistema muy usado en desarrollos de software.

La última forma de llamada es:

```
$ gdb programa pid
```

Esta sentencia lanza el *debugger* y lo asocia a la ejecución de un proceso ya iniciado. El proceso se especifica con el parámetro `pid`. Una vez realizada la llamada al *debugger*, el proceso se detiene y gdb toma el control.

Cuando se ha invocado el intérprete de comandos de db aparece el *prompt*:

```
(gdb)
```

A partir de este momento se pueden ejecutar comandos.

1.1.2. Comandos básicos

Este *debugger* cuenta con una ayuda en línea muy completa. El comando `help` ofrece en todo momento información sobre los comandos disponibles. Ejecutando `help` sin ningún parámetro se obtendrá una ayuda general, donde se muestran las diferentes secciones en las que están divididas los comandos.

```
(gdb) help
List of classes of commands:
aliases -- Aliases of other commands
```

```
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands
Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.

(gdb)
```

Escribiendo `help` seguido de la sección en la que se está interesado, se proporcionará una lista de los comandos disponibles en esa sección. Se puede tener información de cada comando escribiendo el comando `help` más la instrucción:

```
(gdb) help break
Set breakpoint at specified line or function.
break [LOCATION] [thread THREADNUM] [if CONDITION]
LOCATION may be a line number, function name, or "*" and an address.
If a line number is specified, break at start of code for that line.
If a function is specified, break at start of code for that function.
If an address is specified, break at that exact address.
With no LOCATION, uses current execution address of selected stack frame.
This is useful for breaking on return to a stack frame.

THREADNUM is the number from "info threads".
CONDITION is a boolean expression.

Multiple breakpoints at one place are permitted, and useful if conditional.

Do "help breakpoints" for info on other commands dealing with breakpoints.
(gdb)
```

El entorno ofrece la capacidad de autocompletado de comandos mediante el uso de la tecla *TAB*. Así pues, presionando esta tecla mientras se está escribiendo un comando, el sistema intentará completarlo. Si no es capaz, ofrecerá una lista de opciones para que el usuario pueda elegir.

- `run`: ejecución del programa. Este comando ejecuta el programa cargado en memoria. También se puede ejecutar con `r` solamente. Se le pueden pasar parámetros, como se le pasarían al programa que se está analizando, desde el intérprete de comandos.

- `start`: ejecución del programa y parada. Este comando ejecuta el comando cargado en memoria y detiene su ejecución al principio de la ejecución del código principal del programa. En el caso de tratarse de un programa escrito en C, se detendría al principio de la función `main`. Esto solo es válido si el ejecutable cuenta con la tabla de símbolos. En caso contrario, la ejecución del programa continuará hasta el final, devolviendo mensajes de que no se encuentra la tabla de símbolos.

- `continue`: reanudación de ejecución. Este comando reanuda la ejecución del programa después de que esta haya sido detenida con alguno de los métodos que permite gdb. También se puede ejecutar con `c` solamente.

- `step`: ejecuta una instrucción del programa. Ejecuta la instrucción entrando en las llamadas a funciones si es necesario. Este comando está asociado a la ejecución de una instrucción escrita en un lenguaje de alto nivel. Si tan solo se dispone de código ensamblador, no debe utilizarse, pues el resultado puede ser impredecible. También se puede ejecutar este comando con `s`.

- `stepi`: ejecuta una instrucción del programa (asm). Ejecuta la instrucción a bajo nivel, entrando en las llamadas a funciones si es necesario. Este comando está asociado a la ejecución de instrucciones en lenguaje máquina. Es el comando recomendado si no se dispone del código fuente del programa. También se puede ejecutar con `si`.

- `next`: ejecuta una instrucción del programa. Ejecuta la instrucción sin entrar en las llamadas a funciones. Este comando está asociado a la ejecución de una instrucción escrita en un lenguaje de alto nivel. Si tan solo se dispone de código ensamblador, no debe utilizarse, pues el resultado puede ser impredecible. También se puede ejecutar este comando con `n`.

- `nexti`. Ejecuta una instrucción del programa a bajo nivel, entrando en las llamadas a funciones si es necesario. Este comando está asociado a la ejecución de instrucciones en lenguaje máquina. Es el comando recomendado si no se dispone del código fuente del programa. También se puede ejecutar con `ni`.

- `break`: definición de *breakpoints*. Una de las instrucciones más interesantes es `break`. Se utiliza para definir puntos (*breakpoints*) en la línea de ejecución del programa donde se interrumpe la ejecución, dando el control del proceso al usuario del gdb. Una vez se tiene el control se pueden ver/cambiar datos de memoria y registros y después reanudar la ejecución nor-

mal del programa hasta otro *breakpoint* o hasta su finalización. También se pueden ejecutar instrucciones paso a paso, para poder ver cómo evoluciona la ejecución del programa.

```
(gdb) break *0x080482f0
Breakpoint 1 at 0x80482f0
(gdb)
```

En el ejemplo se ha definido un *breakpoint* en una dirección de memoria especificada. Las direcciones de memoria tienen que ir precedidas de un asterisco (`*`). Si el programa que se ejecuta cuenta con su tabla de símbolos, entonces se pueden especificar etiquetas (como nombres de funciones) para definir los diferentes puntos de *break*. Si el programa cuenta con información de *debug*, entonces incluso se tiene acceso al código fuente desde el *debugger*. Esta es una situación bastante improbable en programas en sistemas de producción.

- `info break`: información sobre los *breakpoints* definidos. Este comando devuelve una lista con los *breakpoints* definidos hasta el momento.

```
(gdb) info break
Num      Type       Disp      Enb      Address      What
1      breakpoint   keep       y      0x080483b2   <main+14>
(gdb)
```

- `tbreak`: *breakpoint* temporal. Este comando realiza la misma función que `break`, pero de manera temporal, de tal manera que cuando el *breakpoint* ha sido utilizado una vez, se desactiva automáticamente.

```
(gdb) tbreak *0x080483bc
Breakpoint 2 at 0x80483bc
(gdb) info break
Num      Type       Disp      Enb      Address      What
1      breakpoint   keep       y      0x080483b2   <main+14>
       breakpoint  already    hit         1          time
2      breakpoint    del       y      0x080483bc   <main+24>
(gdb)
```

En el listado aparecen dos *breakpoints*, uno que es permanente (`keep`) y otro que es temporal `(del)`. `Keep` indica que una vez que se haya alcanzado el *breakpoint*, este debe conservarse para posteriores usos. `Del` indica que cuando se alcance el *breakpoint*, este sea borrado.

- `watch`: definición de *watchpoints*. Detiene la ejecución del programa (al igual que los *breakpoints*) cuando el valor apuntado por la expresión definida por el parámetro cambia (escritura de valores). En ese momento, el control del programa es devuelto al usuario mostrándole el valor antiguo y el nuevo valor:

```
(gdb) watch *0xbfbd9960
Hardware watchpoint 3: *3216873824
(gdb) c
Continuing.
Hardware watchpoint 3: *3216873824

Old value = -1208159664
New value = 7
0x080483ce in main ()
```

Existen un par de variantes de *watch:*

- `rwatch`: detiene la ejecución del programa si el valor apuntado es leído.
- `awatch`: detiene la ejecución del programa si el valor apuntado es leído o escrito.
- `delete`: borra un *breakpoint*. Elimina un *breakpoint* de manera definitiva. Hay que indicar el número de *breakpoints* a eliminar, teniendo en cuenta la lista provista por `info break`.

```
(gdb) info break
Num     Type       Disp      Enb      Address      What
1     breakpoint   keep       y     0x080483b2   <main+14>
(gdb) delete 1
(gdb) info break
No breakpoints or watchpoints.
(gdb)
```

- `enable/disable`: activa o desactiva temporalmente un *breakpoint*. Al contrario que la instrucción `delete`, la activación/desactivación de los *breakpoints* es temporal. Se puede observar el estado de estos en `info break`.

```
(gdb) info break
Num     Type       Disp      Enb      Address      What
1     breakpoint   keep       y      0x080483b2  <main+14>
-     breakpoint  already    hit         1         time
2     breakpoint   keep       y      0x080483dc  <main+56>
(gdb) disable 2
(gdb) info break
Num     Type       Disp      Enb      Address      What
1     breakpoint   keep       y      0x080483b2  <main+14>
-     breakpoint  already    hit         1         time
2     breakpoint   keep       n      0x080483dc  <main+56>
(gdb) enable 2
(gdb) info break
Num     Type       Disp      Enb      Address      What
1      breakpoint  keep       y      0x080483b2  <main+14>
-      breakpoint already    hit         1         time
2      breakpoint  keep       y      0x080483dc  <main+56>
(gdb)
```

En el listado se observa cómo el estado (indicado por la columna `Enb`) del *breakpoint* número 2 cambia de `y` a `n`, y después de `n` a `y`. La columna `Enb` indica si el *breakpoint* está `enabled`.

- `ignore`: ignora un número determinado de pasadas sobre un *breakpoint*.

```
gdb) break *0x080483dc
Breakpoint 2 at 0x80483dc
(gdb) info break
Num      Type      Disp      Enb      Address      What
1     breakpoint   keep       y      0x080483b2  <main+14>
      breakpoint  already    hit         1         time
2     breakpoint   keep       y      0x080483dc  <main+56>
(gdb) ignore 2 1
Will ignore next crossing of breakpoint 2.
(gdb) info break
Num      Type      Disp      Enb      Address      What
1     breakpoint   keep       y      0x080483b2  <main+14>
      breakpoint  already    hit         1         time
2     breakpoint   keep       y      0x080483dc  <main+56>
ignore next 1 hits
(gdb) c
Continuing.

Program exited with code 01.
(gdb)
```

En el ejemplo se define el *breakpoint* número 2 y se indica que debe ser ignorado una vez. En el momento de hacer esto, el programa está detenido en el *breakpoint* número 1. Después de hacer la definición del *breakpoint* número 2, se reanuda la ejecución del programa. Este termina debido a que el programa en ejecución no pasa dos veces por el *breakpoint* número 2.

Este tipo de *breakpoint* es útil para avanzar, de manera controlada, en la ejecución de bucles.

- `finish`: continúa la ejecución hasta la finalización de la función actual.

Si el *debugger* se encuentra en una función que puede reconocer y se ejecuta esta instrucción, la ejecución del programa continúa hasta que la función finaliza, devolviendo la ejecución a la línea siguiente de la llamada a la función.

- `display`: activa la visualización constante de un valor.

Este comando crea una visualización continuada de un valor especificado. Cada vez que el programa se detiene se muestra el valor indicado.

```
(gdb) display /1xw 0xbfceea80
1: x/xw 3218008704 0xbfceea80:      0xb7fe2250
(gdb) si
0x080483c9 in main ()
1: x/xw 3218008704 0xbfceea80:      0xb7fe2250
(gdb) si
0x080483cb in main ()
1: x/xw 3218008704 0xbfceea80:      0xb7fe2250
(gdb) si
0x080483ce in main ()
1: x/xw 3218008704 0xbfceea80:      0x00000007
(gdb)
```

En el ejemplo se define un `display` que muestra en todo momento la posición de memoria `0xbfceea80`. La dirección de memoria se especifica como segundo parámetro del comando. El primer comando especifica el formato con el que se mostrarán los datos. La definición del formato tiene que empezar

con una /, a continuación el número de elementos a mostrar, después el tipo de valor a mostrar y finalmente el tamaño de cada elemento mostrado. Los posibles tipos de valores son:

```
o      octal          f      float          x      hexadecimal
d      decimal        u      unsigned deci- a      address
                      mal
t      binary         s      string         c      char
```

El tamaño de los elementos puede ser:

```
b      byte      h      halfword w      word      g      giant
(8 bytes)
```

Así pues, en el ejemplo se especifica que se tiene que mostrar un dato en hexadecimal de tamaño "*word*".

- `info display`: muestra información sobre *displays*. Este comando muestra los *displays* definidos.

```
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
1: y /1wx 3218008704
(gdb)
```

En el ejemplo se muestra que hay un `display` definido y activado. Los *displays* pueden activarse y desactivarse utilizando los comandos `enable/disable display`. El funcionamiento es igual al descrito para el comando `enable/disable` utilizado en los *breakpoints*.

- `undisplay`: borra un *display* definido. Este comando elimina de la lista de *displays* definidos el elemento seleccionado. El siguiente comando borra el *display* definido en el ejemplo anterior.

```
(gdb) undisplay 1
(gdb)
```

- `x`: examinar. Este comando es muy parecido al comando `display` en cuanto a su sintaxis. Consta también de dos parámetros. El primero es el formato y el segundo es el valor a mostrar. En este caso se muestran las posiciones de memoria especificadas y no hay reiteración automática. Este comando se usa para explorar posiciones de memoria.

```
(gdb) x /8xw 0xbfceea80
0xbfceea80: 0x00000007 0xbfceeaa0 0xbfceeaf8 0xb7e73455
0xbfceea90: 0x08048400 0x080482f0 0xbfceeaf8 0xb7e73455
(gdb)
```

- `print`: muestra un dato especificado. Este comando se utiliza para mostrar datos concretos de manera puntual. El comando `x` es muy útil para explorar la memoria, pero para mostrar un dato concreto es más práctico utilizar este comando.

```
(gdb) print /x *0xbf8da640
$4 = 0xb7f64a09
```

El resultado de la ejecución de este comando es la información pedida más una referencia a esta. En el ejemplo se observa un `$4` añadido al principio de la respuesta del comando. Esta referencia puede ser utilizada en otros comandos, para usar el valor que se ha obtenido en comandos `print` anteriores.

```
(gdb) print /x *$4
$6 = 0xb5ebc381
```

- `info registers`: muestra los registros del procesador.

```
(Gdb) Info Registers
eax      0Xbf8F7724      1081116892
ecx      0Xbf8F76A0      1081117024
edx      0X1             1
ebx      0Xb7Fbcff4      1208233996
esp      0Xbf8F7684      0Xbf8F7684
ebp      0Xbf8F7688      0Xbf8F7688
esi      0X8048400       134513664
edi      0X80482F0       134513392
eip      0X80483B2       0X80483B2 <Main+14>
eflags   0X200286        [ Pf Sf If Id ]
cs       0X73            115
ss       0X7B            123
ds       0X7B            123
es       0X7B            123
fs       0X0             0
gs       0X33            51
```

También se puede acceder al valor de los registros utilizando alguno de los comandos anteriores de mostrar información. Se puede hacer referencia a los registros utilizando su nombre genérico y ubicando un `$` al principio. Observemos los siguientes ejemplos:

```
(gdb) print /x $eip
$1 = 0x80483b2
(gdb) x /16xb $eip
0x80483b2 <main+14>: 0x83 0xec 0x24 0xc7 0x45 0xf0 0x03 0x00
0x80483ba <main+22>: 0x00 0x00 0xc7 0x45 0xf4 0x04 0x00 0x00
(gdb) x /8xw $esp
0xbfd1d2b4: 0xbfd1d2d0 0xbfd1d328 0xb7da3455 0x08048400
0xbfd1d2c4: 0x080482f0 0xbfd1d328 0xb7da3455 0x00000001
```

En la primera instrucción se muestra el contenido del registro EIP.

El segundo ejemplo hace un volcado de memoria de los 16 bytes que aparecen a continuación del punto donde apunta el registro EIP.

El tercer ejemplo hace un volcado del contenido de la pila (usando elementos de tamaño *word*) a partir de la dirección apuntada por el registro ESP.

- `info all-registers`: muestra todos los registros. El comando anterior muestra los registros generales, pero el procesador tiene más registros. Se puede ver la lista completa de registros usando este comando.

- `disassemble`: desensambla el código máquina. Este comando desensambla el código en ejecución. Ejecutado sin parámetros, desensambla la función que en ese momento se encuentra en ejecución y es capaz de identificarla. Para que la pueda identificar es necesario que el programa contenga la tabla de símbolos. En el caso de que esta no esté presente, se le pueden indicar un par de direcciones de memoria para que lo desensamble.

```
(gdb) disassemble
Dump of assembler code for function main:
0x080483a4 <main+0>:       lea 0x4(%esp),%ecx
0x080483a8 <main+4>:       and $0xfffffff0,%esp
0x080483ab <main+7>:       pushl -0x4(%ecx)
0x080483ae <main+10>:      push %ebp
0x080483af <main+11>:      mov %esp,%ebp
0x080483b1 <main+13>:      push %ecx
0x080483b2 <main+14>:      sub $0x24,%esp
0x080483b5 <main+17>:      movl $0x3,-0x10(%ebp)
0x080483bc <main+24>:      movl $0x4,-0xc(%ebp)
0x080483c3 <main+31>:      mov -0xc(%ebp),%edx
0x080483c6 <main+34>:      mov -0x10(%ebp),%eax
0x080483c9 <main+37>:      add %edx,%eax
0x080483cb <main+39>:      mov %eax,-0x8(%ebp)
0x080483ce <main+42>:      mov -0x8(%ebp),%eax
0x080483d1 <main+45>:      mov %eax,0x4(%esp)
0x080483d5 <main+49>:      movl $0x80484b0,(%esp)
0x080483dc <main+56>:      call 0x80482d8 <printf@plt>
0x080483e1 <main+61>:      mov $0x1,%eax
0x080483e6 <main+66>:      add $0x24,%esp
0x080483e9 <main+69>:      pop %ecx
0x080483ea <main+70>:      pop %ebp
0x080483eb <main+71>:      lea -0x4(%ecx),%esp
0x080483ee <main+74>:      ret
End of assembler dump.
(gdb)
```

Este ejemplo cuenta con la tabla de símbolos y la ejecución se encontraba en un punto identificado por un elemento de esta, con lo que gdb ha podido identificar la función y la ha desensamblado entera.

```
(no debugging symbols found)
(gdb) break *0x080482f0
Breakpoint 1 at 0x80482f0
(gdb) r
Starting program: ejemplo1_strip
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)


Breakpoint 1, 0x080482f0 in ?? ()
```

```
(gdb) disassemble $eip-20 $eip+30
Dump of assembler code from 0x80482dc to 0x804830e:
0x080482dc <printf@plt+4>:      add $0x8,%al
0x080482de <printf@plt+6>:      push $0x10
0x080482e3 <printf@plt+11>:     jmp 0x80482a8
0x080482e8: add %al,(%eax)
0x080482ea: add %al,(%eax)
0x080482ec: add %al,(%eax)
0x080482ee: add %al,(%eax)
0x080482f0 <printf@plt+24>:     xor %ebp,%ebp
0x080482f2 <printf@plt+26>:     pop %esi
0x080482f3 <printf@plt+27>:     mov %esp,%ecx
0x080482f5 <printf@plt+29>:     and $0xfffffff0,%esp
0x080482f8 <printf@plt+32>:     push %eax
0x080482f9 <printf@plt+33>:     push %esp
0x080482fa <printf@plt+34>:     push %edx
0x080482fb <printf@plt+35>:     push $0x80483f0
0x08048300 <printf@plt+40>:     push $0x8048400
0x08048305 <printf@plt+45>:     push %ecx
0x08048306 <printf@plt+46>:     push %esi
0x08048307 <printf@plt+47>:     push $0x80483a4
0x0804830c.<printf@plt+52>:     call ..0x80482c8 <__libc_start_main@plt>
End of assembler dump.
(gdb)
```

En este caso, el programa no cuenta con la tabla de símbolos y, por lo tanto, el *debugger* no puede identificar funciones automáticamente (como, por ejemplo, la función `main`). Así pues, hay que seguir la ejecución del programa sin prestar mucha atención a la identificación de funciones que el *debugger* intenta hacer. Uno de los datos importantes a conocer para poder ejecutar este programa paso a paso es el `entry point` o `start address` del programa. Para este cometido se usa el comando `info target`.

- `info target`: da información sobre el programa cargado. Una de las informaciones más relevantes es el `entry point` o `start address`.

```
(gdb) info target
Symbols from "ejemplo1_strip".
Local exec file:
ejemplo1_strip', file type elf32-i386.
Entry point: 0x80482f0 0x08048114 - 0x08048127 is .interp
0x08048128 - 0x08048148 is .note.ABI-tag
0x08048148 - 0x08048170 is .hash
0x08048170 - 0x08048190 is .gnu.hash
0x08048190 - 0x080481e0 is .dynsym
0x080481e0 - 0x0804822c is .dynstr
0x0804822c - 0x08048236 is .gnu.version
0x08048238 - 0x08048258 is .gnu.version_r
0x08048258 - 0x08048260 is .rel.dyn
0x08048260 - 0x08048278 is .rel.plt
0x08048278 - 0x080482a8 is .init
0x080482a8 - 0x080482e8 is .plt
0x080482f0 - 0x0804848c is .text
0x0804848c - 0x080484a8 is .fini
0x080484a8 - 0x080484c9 is .rodata
0x080484cc - 0x080484d0 is .eh_frame
0x080494d0 - 0x080494d8 is .ctors
0x080494d8 - 0x080494e0 is .dtors
0x080494e0 - 0x080494e4 is .jcr
0x080494e4 - 0x080495b4 is .dynamic
0x080495b4 - 0x080495b8 is .got
0x080495b8 - 0x080495d0 is .got.plt
0x080495d0 - 0x080495d8 is .data
0x080495d8 - 0x080495e0 is .bss
```

En un programa que no contenga la tabla de símbolos, esta información es importante para poder empezar el *debugging* del programa. El método habitualmente usado es definir un *breakpoint* en la dirección del `entry point` y empezar la sesión de *debugging* a partir de ese punto.

- `set`: cambia el valor de un elemento. Con este comando se puede cambiar el valor de un elemento dado. Cualquier posición de memoria o registro puede ser cambiado de valor.

```
(gdb) x /16xw $esp
0xbf9fdf70: 0xb7f87a09 0x080495b8 0xbf9fdf88 0x080482a4
0xbf9fdf80: 0xb7fc2ff4 0x080495b8 0xbf9fdfa8 0x08048419
0xbf9fdf90: 0xb7ff1250 0xbf9fdfb0 0xbf9fe008 0xb7e82455
0xbf9fdfa0: 0x08048400 0x080482f0 0xbf9fe008 0xb7e82455
(gdb) set {int}($esp+4)=0x0
(gdb) x /16xw $esp
0xbf9fdf70: 0xb7f87a09 0x00000000 0xbf9fdf88 0x080482a4
0xbf9fdf80: 0xb7fc2ff4 0x080495b8 0xbf9fdfa8 0x08048419
0xbf9fdf90: 0xb7ff1250 0xbf9fdfb0 0xbf9fe008 0xb7e82455
0xbf9fdfa0: 0x08048400 0x080482f0 0xbf9fe008 0xb7e82455
```

En este ejemplo se cambia una posición de memoria ubicada en la pila, utilizando como referencia el registro ESP y desplazando el objetivo 4 bytes dentro de la pila. Para poder cambiar el valor se ha tenido que hacer un *typecasting* del valor guardado en el registro ESP. Como en la pila pueden guardarse valores de cualquier tipo, el puntero a la pila es genérico. Para poder escribir un valor es necesario proporcionar un tipo. Este marcará el tamaño de los datos a escribir. En este caso se ha escrito un 0 de tamaño `int` (4 bytes). Para reescribir tan solo un byte se deberá especificar el tipo `char`.

```
(gdb) x /8xw $esp
0xbfba1934: 0xbfba1950 0xbfba19a8 0xb7e26455 0x08048400
0xbfba1944: 0x080482f0 0xbfba19a8 0xb7e26455 0x00000001
(gdb) set {char}($esp+5)=0x0
(gdb) x /8xw $esp
0xbfba1934: 0xbfba1950 0xbfba00a8 0xb7e26455 0x08048400
0xbfba1944: 0x080482f0 0xbfba19a8 0xb7e26455 0x00000001
(gdb)
```

Los registros también se pueden alterar:

```
(gdb) info registers edx
edx 0x1 1
(gdb) set $edx=0xff
(gdb) info registers
edx 0xff 255
```

Incluso el código del programa es alterable. Se puede cambiar el código máquina para alterar la ejecución del mismo. Para hacer esto, es necesario conocer cuáles son los *opcodes* que se quieren poner para reemplazar los existentes. Antes de realizar un cambio de este tipo, hay que documentarse al respecto, para ver si el cambio que se quiere realizar es factible o no.

1.1.3. Ejecución de un programa simple

Para este ejemplo se considera un programa que cuando se ejecuta sin parámetros devuelve el siguiente resultado:

```
$ ejemplo1
Resultado de x + y = 7
$
```

Este programa es el ejemplo que ya se ha tratado anteriormente. Se realizará el trazado suponiendo dos situaciones distintas:

- El ejecutable cuenta con la tabla de símbolos.
- El ejecutable no tiene la tabla de símbolos.

Si un programa conserva su tabla de símbolos, hacer una sesión de *debug* con él resulta mucho más sencillo.

1.1.4. Ejecutable con tabla de símbolos

Primeramente se carga el *debugger* con el programa que se quiere tratar:

```
$gdb ejemplo1
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
```

```
This GDB was configured as "i486-linux-gnu"...
(gdb)
```

Una vez en el entorno de ejecución, no se recibe ningún mensaje que indique que el programa no tiene tabla de símbolos. Se puede, por lo tanto, intentar avanzar hasta la función principal del programa.

```
(gdb) start
Breakpoint 1 at 0x80483b2
Starting program: /home/jgay/projectes/uoc/bin/ejemplo1_nodebug
0x080483b2 in main ()
Current language: auto; currently asm
(gdb)
```

El *debugger* consigue arrancar el programa y se tiene la ejecución en la función main que detecta sin ningún problema. Una vez en esta función, se puede desensamblar la función automáticamente:

```
(gdb) disassemble
Dump of assembler code for function main:
0x080483a4 <main+0>:        lea 0x4(%esp),%ecx
0x080483a8 <main+4>:        and $0xfffffff0,%esp
0x080483ab <main+7>:        pushl -0x4(%ecx)
0x080483ae <main+10>:       push %ebp
0x080483af <main+11>:       mov %esp,%ebp
0x080483b1 <main+13>:       push %ecx
0x080483b2 <main+14>:       sub $0x24,%esp
0x080483b5 <main+17>:       movl $0x3,-0x10(%ebp)
0x080483bc <main+24>:       movl $0x4,-0xc(%ebp)
0x080483c3 <main+31>:       mov -0xc(%ebp),%edx
0x080483c6 <main+34>:       mov -0x10(%ebp),%eax
0x080483c9 <main+37>:       add %edx,%eax
0x080483cb <main+39>:       mov %eax,-0x8(%ebp)
0x080483ce <main+42>:       mov -0x8(%ebp),%eax
0x080483d1 <main+45>:       mov %eax,0x4(%esp)
0x080483d5 <main+49>:       movl $0x80484b0,(%esp)
0x080483dc <main+56>:       call 0x80482d8 <printf@plt>
0x080483e1 <main+61>:       mov $0x1,%eax
0x080483e6 <main+66>:       add $0x24,%esp
0x080483e9 <main+69>:       pop %ecx
0x080483ea <main+70>:       pop %ebp
0x080483eb <main+71>:       lea -0x4(%ecx),%esp
0x080483ee <main+74>:       ret
End of assembler dump.
(gdb)
```

Este es el código ensamblador de la función main. Se puede observar la llamada a la función `printf`; teniendo en cuenta que el programa escribe una cadena de caracteres como salida, esta tiene que ser la llamada que realiza dicha operación. Justo antes de la llamada `call 0x80482d8 <printf@plt>` se tienen que preparar los parámetros en la pila para que la llamada a la función tenga éxito. El primer parámetro se introduce en la pila mediante la sentencia previa a la instrucción `call`:

```
movl $0x80484b0,(%esp)
```

Con la instrucción anterior se introduce en la parte superior de la pila la dirección de memoria indicada. Se examina entonces el contenido de esa dirección de memoria:

```
(gdb) x /16cb 0x80484b0
0x80484b0: 82 'R' 101 'e' 115 's' 117 'u' 108 'l' 116 't' 97 'a' 100 'd'
0x80484b8: 111 'o' 32 ' ' 100 'd' 101 'e' 32 ' ' 120 'x' 32 ' ' 43 '+'
(gdb) x /1s 0x80484b0
0x80484b0: "Resultado de x + y = %d\n"
(gdb)
```

Se ve que la dirección de memoria contiene un puntero al *string* que servirá de formato para imprimir el resultado por pantalla. De esta cadena de caracteres se deduce que falta un segundo parámetro, que será un entero e indicará el valor del resultado. El segundo parámetro se introduce en la pila con la sentencia anterior:

```
mov %eax,0x4(%esp)
```

Dado que la ejecución del programa aún no ha alcanzado ese punto, ver cuál es el valor de este parámetro daría como resultado un valor equivocado. Se avanza, entonces, la ejecución del programa hasta justo antes de realizar la llamada a la función `printf`, utilizando la instrucción `ni (nexti)`.

```
(gdb) info program

Using the running image of child process 17060.
Program stopped at 0x80483dc.
It stopped after being stepped.
(gdb)
```

En este punto ya se puede ver cuál es el valor que se va a mostrar por pantalla:

```
(gdb) print $eax
$1 = 7
(gdb)
```

Se ha consultado el valor del registro EAX, pero también se puede observar este valor en la pila, porque ya está preparado para ser pasado como parámetro a la función llamada.

```
(gdb) x /1xw $esp+4
0xbfb808f4: 0x00000007
(gdb)
```

A modo de ejemplo, pasamos a alterar el contenido de la pila para que el valor mostrado, cuando se ejecute la llamada a `printf`, sea otro, por ejemplo `123`:

```
(gdb) set {int}($esp+4)=123
(gdb) x /1xw $esp+4
0xbfb808f4: 0x0000007b
(gdb) x /1dw $esp+4
0xbfb808f4: 123
(gdb)
```

Puesto que ha sido el contenido del registro EAX el que se ha introducido en la pila, para alterar el comportamiento del programa de manera coherente se debería cambiar también el contenido de este registro al nuevo valor:

```
(gdb) set $eax=123
(gdb) print $eax
$2 = 123
(gdb)
```

Una vez realizadas las alteraciones deseadas, se procede a avanzar la ejecución del programa ejecutando la llamada a la función `printf`:

```
(gdb) ni
Resultado de x + y = 123
0x080483e1 in main ()
(gdb)
```

Se observa que el resultado del programa ha sido alterado correctamente. A partir de este punto se ejecuta el programa normalmente hasta su finalización.

```
(gdb) continue
Continuing.
Program exited with code 01.
(gdb)
```

Se ha visto que la ejecución de un programa con su tabla de símbolos es relativamente sencilla, puesto que no es necesario localizar los diferentes elementos que se ejecutan, ya que el *debugger* es capaz de hacerlo por sí solo.

1.1.5. Ejecutable sin tabla de símbolos

Al igual que en el caso anterior, se carga el *debugger* con el programa que se quiere tratar:

```
$ gdb ejemplo1
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(no debugging symbols found)
(gdb)
```

En este caso el *debugger* da un aviso de que no hay símbolos disponibles; esto significa que gdb no puede localizar automáticamente las distintas secciones del programa y que, por lo tanto, la localización de la función principal main deberá hacerse manualmente. Se procede a realizar la misma manipulación que en el caso anterior, o sea, que el programa devuelva 123 en lugar de 7. Para comprobar que el *debugger* no puede realizar la detección de la función principal, ejecutamos la instrucción `start`:

```
(gdb) start
Function "main" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (main) pending.
Starting program: /home/jgay/projectes/uoc/bin/ejemplo1_strip
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
Resultado de x + y = 7

Program exited with code 01.
(gdb)
```

Al ejecutar la instrucción, el *debugger* no encuentra el símbolo pedido (`main`) y pregunta si puede ser que este símbolo se encuentra en alguna librería que se cargará más adelante. Se le dice que sí para que lo intente, pero el programa se ejecuta en su totalidad sin detenerse, puesto que el símbolo pedido no existe. El siguiente paso consiste en determinar el `entry point` del programa y detener la ejecución en ese punto.

```
(gdb) info target
Symbols from "ejemplo1".
Local exec file:
`ejemplo1', file type elf32-i386.
Entry point: 0x80482f0      0x08048114 - 0x08048127 is .interp
                            0x08048128 - 0x08048148 is .note.ABI-tag
                            0x08048148 - 0x08048170 is .hash
...
```

Una vez encontrada la dirección de entrada al programa, se define un *breakpoint* en esa dirección y se ejecuta el programa:

```
(gdb) break *0x80482f0
Breakpoint 1 at 0x80482f0
(gdb) info breakpoints
Num      Type      Disp      Enb      Address      What
 1     breakpoint  keep       y     0x080482f0 <printf@plt+24>
(gdb) run
Starting program: /home/jgay/projectes/uoc/bin/ejemplo1_strip
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)

Breakpoint 1, 0x080482f0 in ?? ()
(gdb)
```

Una vez en este punto, se desensambla el código que se ejecutará. Hay que tener en cuenta que el *debugger* no reconoce ninguna función en este punto (`Breakpoint 1, 0x080482f0 in ()`) indicándolo con dos signos de interrogación. Para obtener un desensamblado del código que se ejecutará hay que indicar el intervalo de memoria que se quiere desensamblar:

```
(gdb) disassemble
No function contains program counter for selected frame.
(gdb) disassemble $eip $eip+35
Dump of assembler code from 0x80482f0 to 0x8048313:
0x080482f0 <printf@plt+24>:      xor %ebp,%ebp
0x080482f2 <printf@plt+26>:      pop %esi
0x080482f3 <printf@plt+27>:      mov %esp,%ecx
0x080482f5 <printf@plt+29>:      and $0xfffffff0,%esp
0x080482f8 <printf@plt+32>:      push %eax
0x080482f9 <printf@plt+33>:      push %esp
0x080482fa <printf@plt+34>:      push %edx
0x080482fb <printf@plt+35>:      push $0x80483f0
0x08048300 <printf@plt+40>:      push $0x8048400
0x08048305 <printf@plt+45>:      push %ecx
0x08048306 <printf@plt+46>:      push %esi
0x08048307 <printf@plt+47>:      push $0x80483a4
0x0804830c <printf@plt+52>:      call 0x80482c8 <__libc_start_main@plt>
0x08048311 <printf@plt+57>:      hlt
0x08048312 <printf@plt+58>:      nop
End of assembler dump.
(gdb)
```

Una posible manera de proceder sería colocando un *breakpoint* en cada una de las direcciones de ejecución del programa, que se introducen en la pila. Habitualmente, la última dirección introducida en la pila es la dirección de la función principal main. Si esta información no se considera interesante y se quiere ir directamente a realizar la manipulación querida, entonces se puede proceder de otra manera.

Como se ve que el programa retorna un *string* dando un mensaje, se puede ver cuáles son las funciones del sistema que este programa llama para volcar por pantalla la información.

```
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x080482b8 __gmon_start__@plt
0x080482c8 __libc_start_main@plt
0x080482d8 printf@plt
```

Se observa que el programa tan solo tiene 3 funciones definidas en la sección PLT (*Procedure Linkage Table*). Estas 3 funciones se llaman desde el programa. Así pues, la que interesa para poder hacer la manipulación deseada es `printf`. La dirección que se especifica es el punto de entrada a la llamada a `printf`. Entonces se puede poner un *brekpoint* en esta dirección.

```
(gdb) break *0x080482d8
Breakpoint 2 at 0x80482d8
(gdb) info breakpoints
Num      Type      Disp      Enb      Address      What
 1     breakpoint  keep       y     0x080482f0  <printf@plt+24>
 2     breakpoint  keep       y     0x080482d8  <printf@plt>
(gdb)
```

Se puede continuar con la ejecución del programa y este se detendrá cuando la función `printf` sea llamada. En ese punto se podrá, seguramente, alterar el programa:

```
(gdb) disassemble $eip $eip+20
Dump of assembler code from 0x80482d8 to 0x80482ec:
0x080482d8 <printf@plt+0>:      jmp *0x80495cc
0x080482de <printf@plt+6>:      push $0x10
0x080482e3 <printf@plt+11>:     jmp 0x80482a8
0x080482e8: add %al,(%eax)
0x080482ea: add %al,(%eax)
End of assembler dump.
(gdb)
```

La ejecución se ha detenido justo cuando la llamada a la función `printf` ha sido hecha. Esto significa que el último elemento de la pila indica la dirección de retorno de la llamada, o sea, una dirección que se encuentra en el hilo de ejecución principal del programa. También se conoce que los registros no se han visto alterados a excepción de los registros EIP (la instrucción a ejecutarse ha cambiado) y ESP (se ha empilado la instrucción de retorno).

```
(gdb) x /8xw $esp
0xbfe3539c: <b>0x080483e1</b> 0x080484b0 0x00000007 0xbfe353b8
0xbfe353ac: 0x080482a4 0xb7efbff4 0x080495b8 0x00000003
(gdb) disassemble {char*}$esp-50 ({char*}$esp)+15
Dump of assembler code from 0x80483af to 0x80483f0:
0x080483af <printf@plt+215>:      mov %esp,%ebp
0x080483b1 <printf@plt+217>:      push %ecx
0x080483b2 <printf@plt+218>:      sub $0x24,%esp
0x080483b5 <printf@plt+221>:      movl $0x3,-0x10(%ebp)
0x080483bc <printf@plt+228>:      movl $0x4,-0xc(%ebp)
0x080483c3 <printf@plt+235>:      mov -0xc(%ebp),%edx
0x080483c6 <printf@plt+238>:      mov -0x10(%ebp),%eax
0x080483c9 <printf@plt+241>:      add %edx,%eax
0x080483cb <printf@plt+243>:      mov %eax,-0x8(%ebp)
0x080483ce <printf@plt+246>:      mov -0x8(%ebp),%eax
0x080483d1 <printf@plt+249>:      mov %eax,0x4(%esp)
0x080483d5 <printf@plt+253>:      movl $0x80484b0,(%esp)
0x080483dc <printf@plt+260>:      call 0x80482d8 <printf@plt>
0x080483e1 <printf@plt+265>:      mov $0x1,%eax
0x080483e6 <printf@plt+270>:      add $0x24,%esp
0x080483e9 <printf@plt+273>:      pop %ecx
0x080483ea <printf@plt+274>:      pop %ebp
0x080483eb <printf@plt+275>:      lea -0x4(%ecx),%esp
0x080483ee <printf@plt+278>:      ret
0x080483ef <printf@plt+279>:      nop
End of assembler dump.
(gdb)
```

La llamada a la instrucción `disassemble` se hace utilizando directamente el registro ESP, puesto que contiene la dirección de memoria de un puntero que a su vez contiene la dirección de retorno de la llamada a `printf`. Otra manera de hacer la llamada sería especificando de manera explícita la dirección en la que se basa el desensamblado.

```
(gdb) disassemble 0x080483e1-50 0x080483e1+15
Dump of assembler code from 0x80483af to 0x80483f0:
0x080483af <printf@plt+215>:      mov %esp,%ebp
0x080483b1 <printf@plt+217>:      push %ecx
0x080483b2 <printf@plt+218>:      sub $0x24,%esp
...
```

Una vez que se llega a este punto, se pueden hacer las mismas modificaciones al programa que en el caso anterior, pero teniendo en cuenta que la ejecución del programa se encuentra en otro punto. Así pues, el parámetro buscado se encontrará 4 bytes más adelante teniendo en cuenta que se ha empilado la dirección de retorno.

```
(gdb) x /16xw $esp
0xbfe3539c: 0x080483e1 0x080484b0 0x00000007 0xbfe353b8
0xbfe353ac: 0x080482a4 0xb7efbff4 0x080495b8 0x00000003
0xbfe353bc: 0x00000004 0x00000007 0xbfe353e0 0xbfe35438
0xbfe353cc: 0xb7dbb455 0x08048400 0x080482f0 0xbfe35438
(gdb) set {int}($esp+8)=123
(gdb) x /16xw $esp
0xbfe3539c: 0x080483e1 0x080484b0 0x0000007b 0xbfe353b8
0xbfe353ac: 0x080482a4 0xb7efbff4 0x080495b8 0x00000003
0xbfe353bc: 0x00000004 0x00000007 0xbfe353e0 0xbfe35438
0xbfe353cc: 0xb7dbb455 0x08048400 0x080482f0 0xbfe35438
(gdb) x /1dw $esp+8
0xbfe353a4: 123
(gdb) info registers $eax
eax    0x7    7
(gdb) set $eax=123
(gdb) info registers $eax
eax    0x7b    123
(gdb)
```

Se han modificado los antiguos valores tanto en la pila como en el registro EAX. Una vez hecho esto, ya se puede proceder con la ejecución del programa hasta el final para ver si el cambio ha sido correcto.

```
(gdb) continue
Continuing.
Resultado de x + y = 123

Program exited with code 01.
(gdb)
```

Todo termina correctamente, mostrando el nuevo resultado.

En estos casos en los que la tabla de símbolos ha sido borrada es muy útil una herramienta que complementa a gdb y que se llama `objdump`. Esta herramienta permite mostrar una gran cantidad de información de un fichero ejecutable. Entre sus características podemos encontrar:

- Desensamblado del programa.
- Muestra la tabla de símbolos.
- Muestra datos sobre las distintas secciones del ejecutable.

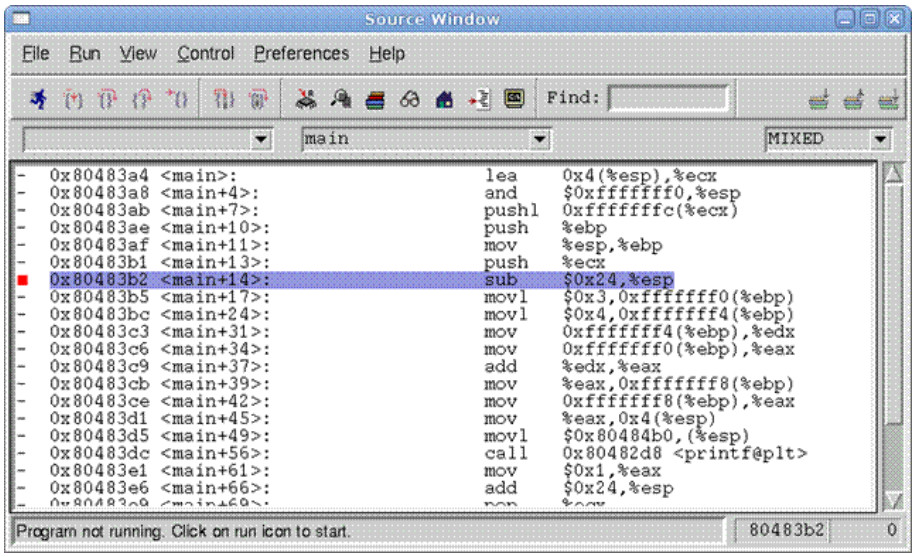
Es muy útil para hacer un análisis separado del *debugger*. Debido a que ofrece listados de una forma sencilla, es una herramienta muy usada para analizar código.

1.1.6. *Insight*

El *debugger* gdb es una herramienta muy potente, pero no es muy intuitiva. Para mejorar su utilización, el *debugger* incorpora una pequeña interfaz en modo texto llamado TUI (*Text User Interface*). Aun siendo una opción interesante, hay otra posibilidad más cómoda. Existe una interfaz gráfica para gdb llamada

(3)**Insight**. Disponible en: http://sources.redhat.com/insight/

Insight[3]. Desde esta interfaz se tiene disponibilidad de toda la potencia de gdb pero de una manera más cómoda. La vista general de Insight cuando arranca es la siguiente:



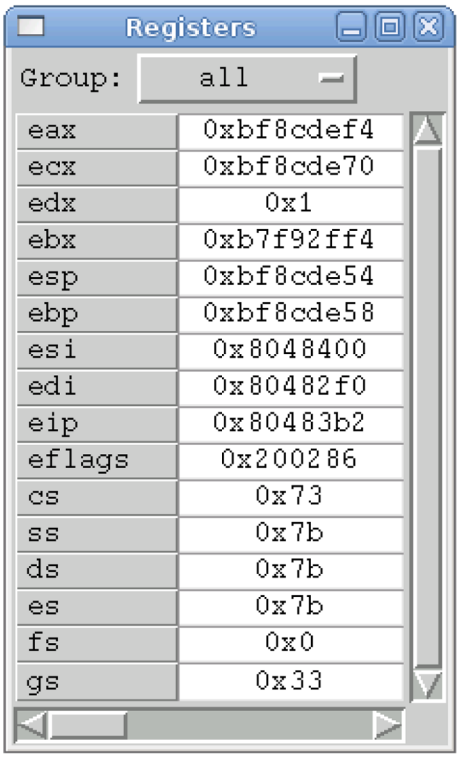
Insight: Ventana del código fuente

En esta ventana se puede observar el código en ensamblador del programa a ejecutar y un *breakpoint* creado automáticamente por el entorno (esto es configurable) en la entrada de la función principal del programa. Hay que tener en cuenta que el *debugger* solo será capaz de ofrecer el entorno en estas condiciones si el programa que se está tratando contiene la tabla de símbolos. En el caso de que no sea así, no se mostrará código alguno a la entrada del programa. Aun así, las diferentes opciones del programa hacen que sea práctico trabajar con él.

La ventana principal permite la inserción de *breakpoints*, tanto fijos como temporales, usando el botón derecho del ratón.

El entorno tiene una serie de ventanas auxiliares que son de ayuda durante el proceso de *debug* de un programa. Entre las más interesantes se encuentran:

- **Ventana de registros**. Esta ventana visualiza en todo momento los registros del procesador.



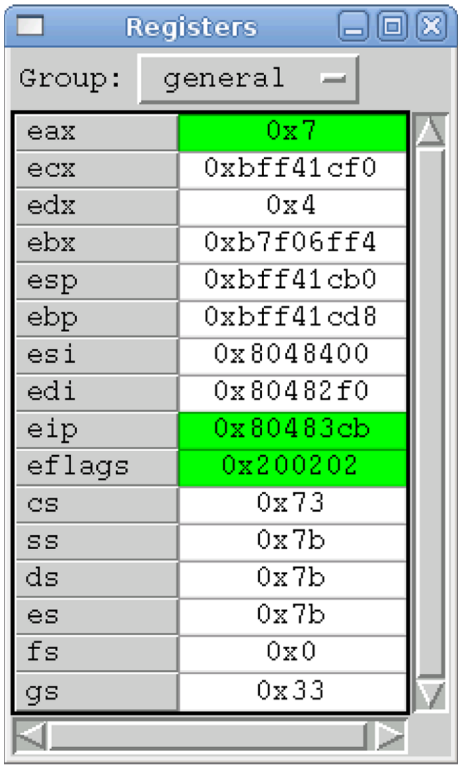
Insight: Ventana de registros

Se puede escoger el subconjunto de registros visualizados. Normalmente el conjunto de registros más útil es el de los registros generales, pero las posibilidades incluyen otros conjuntos:

- `sse: xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, xmm7, mxcsr`
- `mmx: mm0, mm1, mm2, mm3, mm4, mm5, mm6, mm7`
- `general: eax, ecx, edx, ebx, esp, ebp, esi, esi, eip, eflags, cs, ss, ds, es, fs, gs`
- `float: st0, st1, st2, st3, st4, st5, st6, st7, fctrl, fstat, ftag, fiseg, fioff, foseg, fooff, fop`
- `all`: Agrupa los registros de todas las distintas categorías
- `vector`: Agrupa los registros de la categoría `sse` y `mmx`
- `system: orig_eax`

Cada uno de estos conjuntos de registros se usa para una tarea específica. Los registros en la categoría "general" son los más utilizados y los que resulta más útil tener controlados en todo momento.

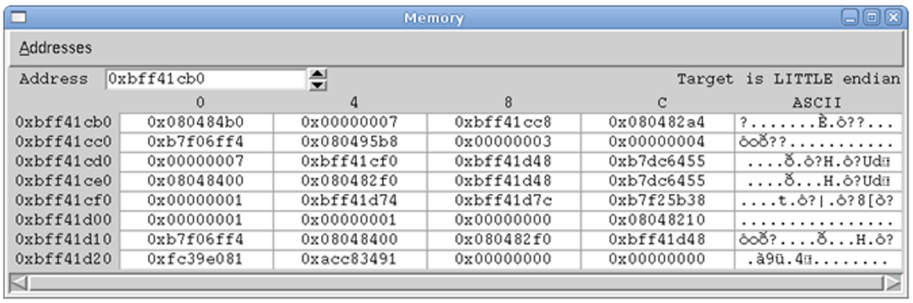
Una función muy interesante de esta ventana es que cada vez que uno de los registros visualizados cambia de valor, el sistema lo cambia de color para indicar que ha sido modificado. Esto es muy interesante cuando se está ejecutando un programa.



Insight: Ventana de registros (cambios)

También se puede cambiar el contenido de un registro situando el cursor sobre la casilla del registro correspondiente e introduciendo un valor nuevo.

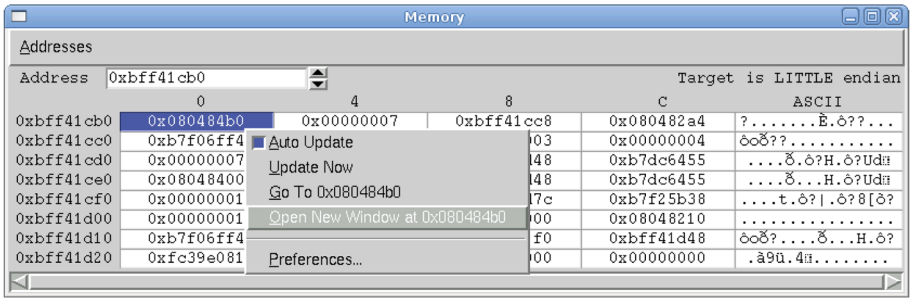
- **Ventana de volcado de memoria**. En esta ventana se inspecciona una zona de la memoria. El número y el tamaño de los elementos a visualizar es configurable.



Insight: Ventana de memoria

Se pueden abrir tantas ventanas de este tipo como se quiera para explorar distintas posiciones de memoria en cada una. Se puede presionar el botón derecho del ratón sobre un registro (por ejemplo, el `esp`) y seleccionar la opción "*Open memory window*". Aparecerá una ventana como la anterior mostrando la dirección seleccionada en la primera dirección de memoria del bloque.

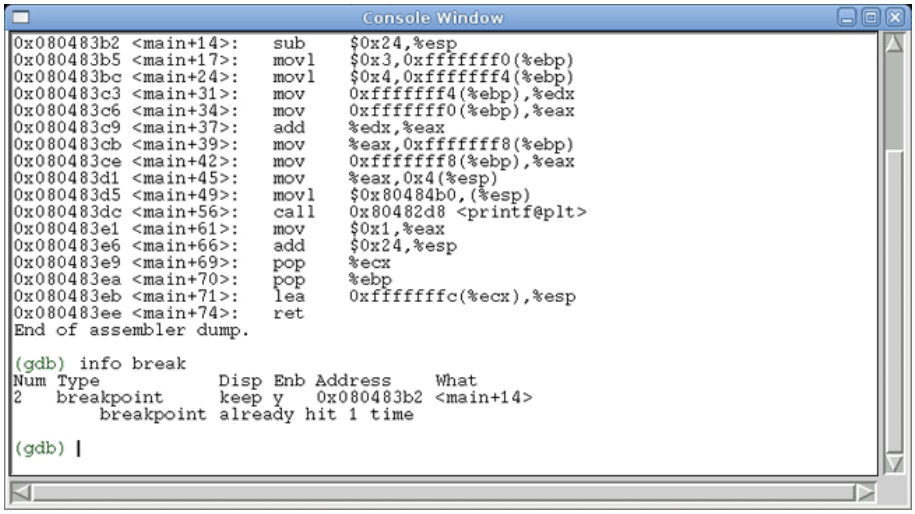
Además, cada posición de memoria es editable y se puede variar el valor de cada celda tan solo seleccionando la celda y cambiando el valor. Presionando el botón derecho del ratón sobre una celda, se puede seleccionar la opción "*Open new window at xxxx*", abriéndose una nueva ventana que explorará una nueva región de memoria.



Insight: Abrir nueva ventana de memoria

El resultado será una nueva ventana mostrando el contenido de la nueva posición de memoria elegida.

- **Ventana de consola**. La ventana de consola ofrece el intérprete de comandos de gdb, como ya se ha visto anteriormente. Todos los comandos del *debugger* quedan a disposición del usuario, que dispone además de las ventajas del entorno gráfico.

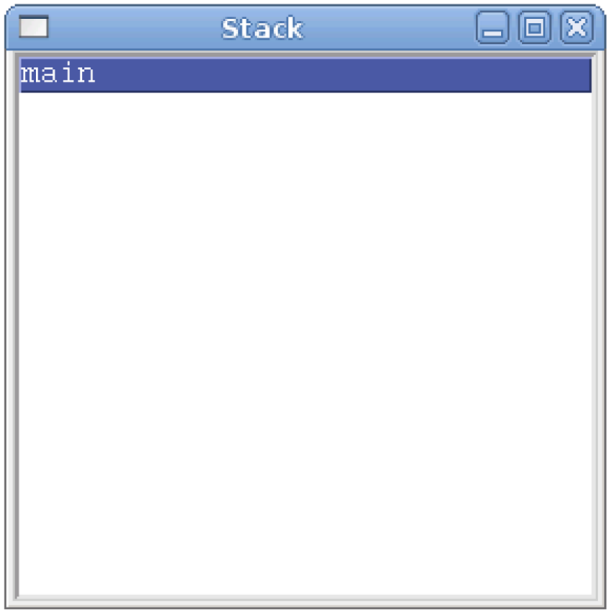


Insight: Ventana de consola

En el caso de que no se disponga de la tabla de símbolos en el ejecutable, la ventana de consola se convierte en la ventana principal durante la sesión de *debugging*. Las ventanas de registros y de memoria se usan para hacer un seguimiento del estado del programa en ejecución. Si se dispone de la tabla de símbolos, entonces la ventana del código fuente se vuelve más útil.

- **Ventana con la lista de las funciones llamadas (*frame stack*)**. Esta ventana resulta útil si se dispone de la tabla de símbolos; si no, no es de utilidad. En ella se muestra la secuencia de llamadas entre funciones. Se puede

cambiar de una función a otra en la lista, pudiéndose consultar información del estado de cada una de las llamadas.



Insight: Lista de funciones llamadas

Aunque *Insight* es una herramienta interesante, no debe olvidarse que el motor de todo este proceso es el *debugger* gdb. El entorno gráfico no deja de ser un añadido a una herramienta que ya de por sí es totalmente útil.

1.1.7. Conclusiones

El *debugger* es una herramienta muy potente. Permite analizar a bajo nivel todo tipo de estructuras ejecutadas, consultadas y modificadas por un programa durante su ejecución. Aunque no se han visto todas las opciones posibles, se ha dado una idea básica de su funcionamiento para ayudar a depurar programas. Esta depuración a bajo nivel es muy útil cuando se están escribiendo *exploits*, tanto para analizar el punto en el cual es (o puede ser) vulnerable un programa como para ver si dicho *exploit* realiza correctamente los cambios que se desean en el programa víctima. En el proceso de aprendizaje de la utilización de esta herramienta también se entra en contacto con elementos del sistema que normalmente pasan desapercibidos para un usuario normal.

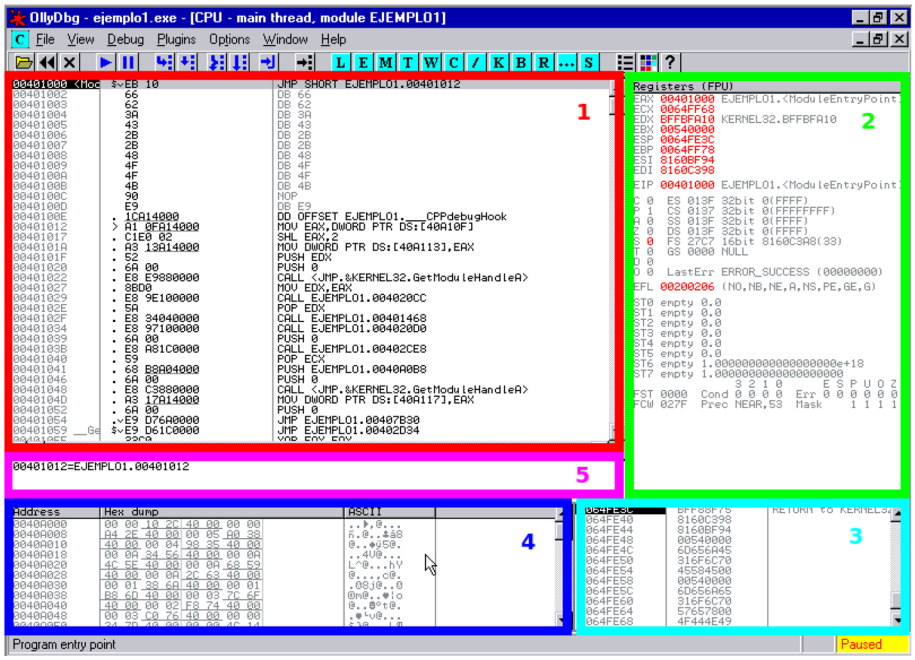
1.2. OllyDbg

OllyDbg es un *debugger* muy utilizado en entornos Windows, aunque no es el único. Microsoft tiene su propia herramienta, WinDbg, e incluso gdb tiene una versión para los sistemas operativos de Microsoft. Conviene señalar que los conceptos para seguir la ejecución de programas a bajo nivel no cambian, ya que tan solo se está cambiando el software, no el hardware. Por consiguiente, se siguen teniendo los mismos registros, la inspección de memoria sigue siendo un punto interesante y el desensamblado de las diferentes instruccio-

nes de código máquina será equivalente. La única manera de que hubiera un cambio grande en la manera de realizar el análisis sería que se tuviera un cambio de arquitectura hardware.

1.2.1. Primer contacto

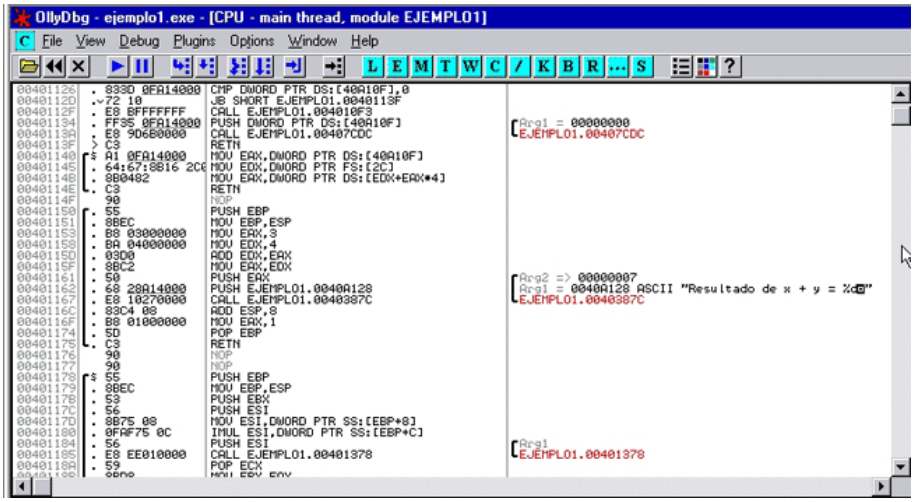
OllyDbg es una aplicación que tiene una ventana principal en la que se van abriendo otras subventanas que dan acceso a diferentes opciones o funcionalidades de la aplicación. Si se abre el *debugger* sin cargar ninguna aplicación, tan solo se obtendrá una ventana vacía, con el menú y unos iconos para poder utilizar. Si en ese momento se abre una aplicación aparecerá una ventana como la del ejemplo:



OllyDbg: Ventana principal

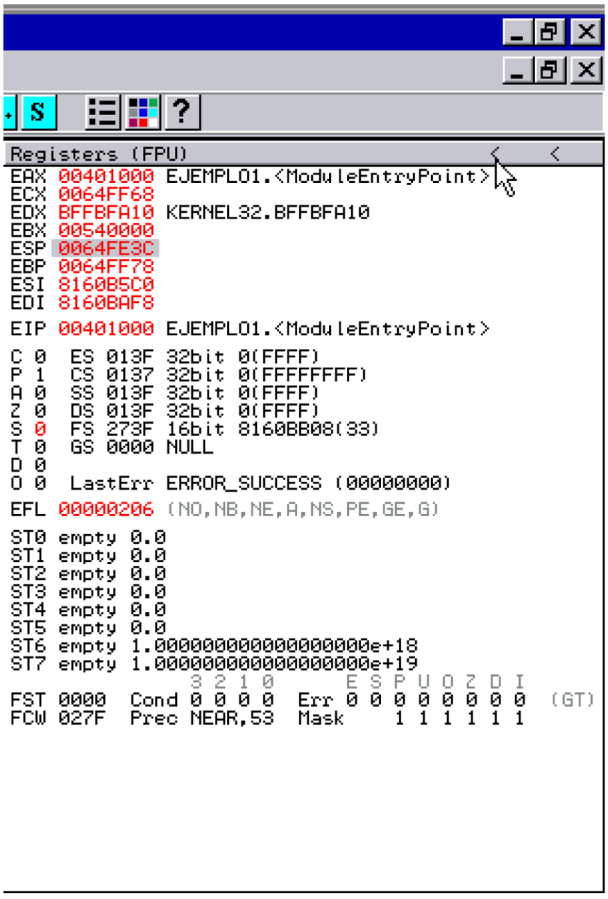
Esta es la ventana principal para el análisis de aplicaciones de OllyDbg. También se denomina "CPU Window". Se puede llamar también con la combinación de teclas "Alt+C". En esta ventana se observan 5 secciones:

1. Desensamblado. En esta zona aparece la dirección de memoria de cada instrucción, el código máquina correspondiente, su traducción a lenguaje ensamblador y una última parte en la que se encuentra el análisis que OllyDbg ha realizado de la aplicación cargada. Se puede ver el análisis del código hecho por OllyDbg en la siguiente figura.



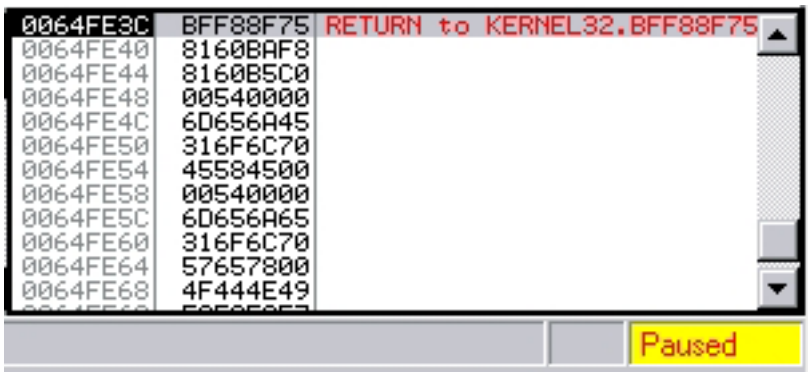
OllyBdg – Sección de desensamblado de código

2. Registros. Es esta sección se puede observar el estado de los registros en cada paso de la ejecución del programa. Cada vez que el programa avanza en la ejecución de una o varias líneas de código, el estado de los registros se actualiza.



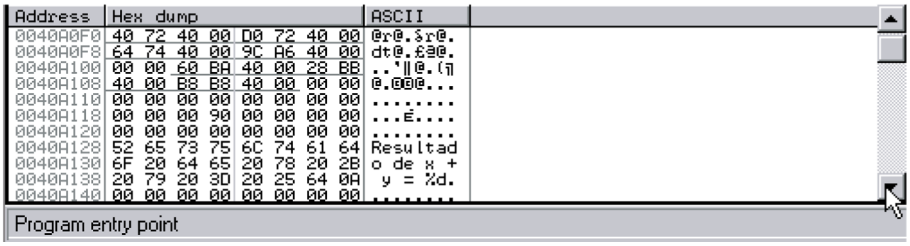
OllyDbg - Sección de registros

3. Pila. Al igual que el resto de secciones, esta también se actualiza a cada paso de la ejecución. En esta sección se muestra el contenido de la pila. Habitualmente, se muestra también un análisis del contenido de la pila en una columna situada a la derecha. En ella se pueden ver fácilmente las direcciones de retorno de las llamadas a funciones hechas con call.



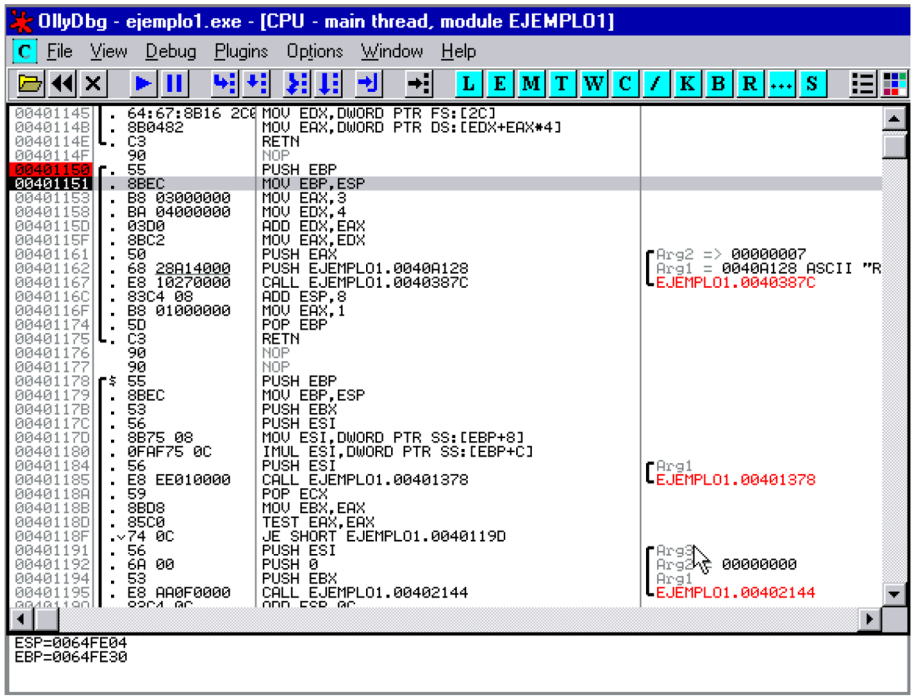
OllyDbg - Sección de contenido de la pila

4. Volcado de memoria. En esta sección se puede ver un volcado del contenido de la memoria. Es posible desplazarse por el volcado e incluso fijar el punto de la memoria que quiere observarse.



OllyDbg – Sección de volcado de memoria

5. Área de ayuda. En esta sección se ofrece información sobre lo que los comandos en ensamblador realizan. Datos sobre los operandos del comando a ejecutar son mostrados siempre que sea posible. Esto facilita la localización de la información en un punto de la ejecución, ya que no tienen que buscarse cada uno de los operandos en las diferentes ventanas de información. Los datos más relevantes en un momento concreto pueden verse aquí, y si se requiere más información, se puede observar el contenido de las otras ventanas.



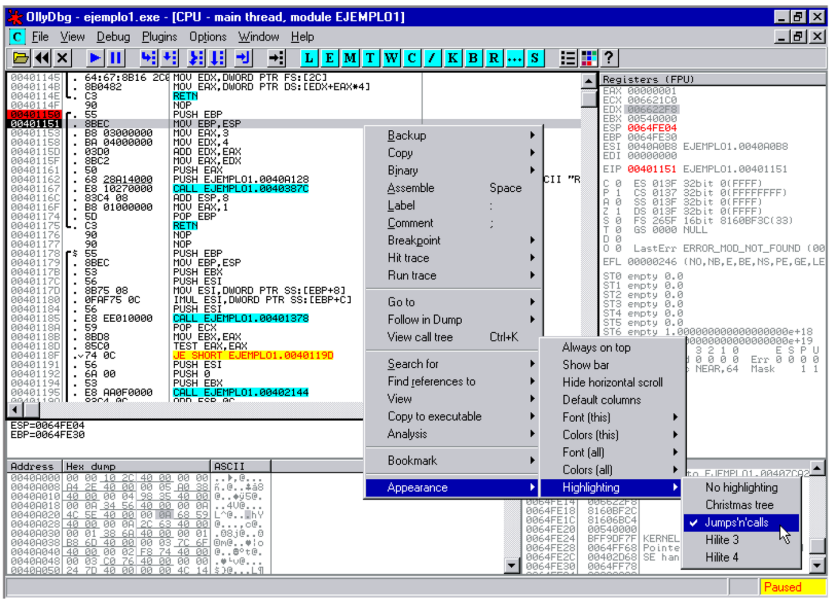
OllyDbg - Desensamblado más información

En la figura anterior puede observarse que se muestra información sobre los registros que van a intervenir en la ejecución de la siguiente instrucción en ensamblador. La instrucción es `MOV EBP,ESP`, y en la parte inferior se puede ver el contenido de ambos registros justo antes de ejecutarse la instrucción.

1.2.2. Opciones

Esta aplicación es muy configurable. Es posible cambiar el tamaño de todas las secciones, además del tipo de letra y coloreado de código.

> Por ejemplo, uno de las configuraciones de coloreado de código que existen es la de marcar las instrucciones relacionadas con llamadas a funciones y saltos. De esta manera, la lectura del código se vuelve más visual.



OllyDbg - Opciones de la sección de código (coloreado)

En la figura anterior se puede observar cómo se pueden configurar las opciones para mostrar datos o código. Se puede apreciar que la ventana del código ensamblador tiene muchas opciones. De hecho, se abre un menú contextual haciendo clic con el botón derecho del ratón en las diferentes secciones de la aplicación. Cada sección cuenta con sus propias opciones y las opciones mostradas dependen en todo momento del elemento seleccionado de la sección. Entre las opciones más interesantes que se pueden encontrar en cada sección hay:

- Desensamblado
  - Modificación del código máquina.
  - Ensamblado de instrucciones a código máquina.
  - Gestión de *breakpoints*.
  - Mostrar datos de una dirección de memoria seleccionada en la sección de volcado de memoria.
  - Ir a un punto de la ejecución del programa.
  - Ver el árbol de llamadas.
  - Buscar un dato concreto.

- Registros
  - Posibilidad de modificar el contenido de todos los registros.
  - Visualización del grupo de registros deseado.

- Pila
  - Posibilidad de modificar los datos de la pila.
  - Alterar la pila introduciendo o sacando elementos (`PUSH/POP`).
  - Mostrar datos de una dirección de memoria seleccionada en la sección de volcado de memoria o en la pila.

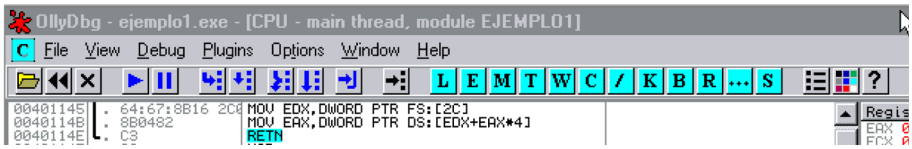
- Volcado de memoria
  - Desensamblado de posiciones de memoria.

- Buscar referencias a datos.
- Gestión de *breakpoints* asociados al acceso a datos.
- Mostrar los datos en diferentes formatos (hexadecimal, ASCII, tamaño de los datos que se visualizan...).

Estas no son todas las opciones, pero son algunas de las más interesantes.

El desensamblado de posiciones de memoria permite comprobar si un código malicioso se ha introducido en memoria de manera correcta para que el *exploit* tenga éxito.

Se dispone también de operaciones para controlar la ejecución del proceso que se está analizando. De la misma manera que gdb contaba con una instrucción run para arrancar el programa y `stepi` para ejecutar una instrucción en ensamblador, en OllyDbg existen opciones parecidas.



OllyDbg - Iconos

Teniendo en cuenta los iconos que aparecen de izquierda a derecha:

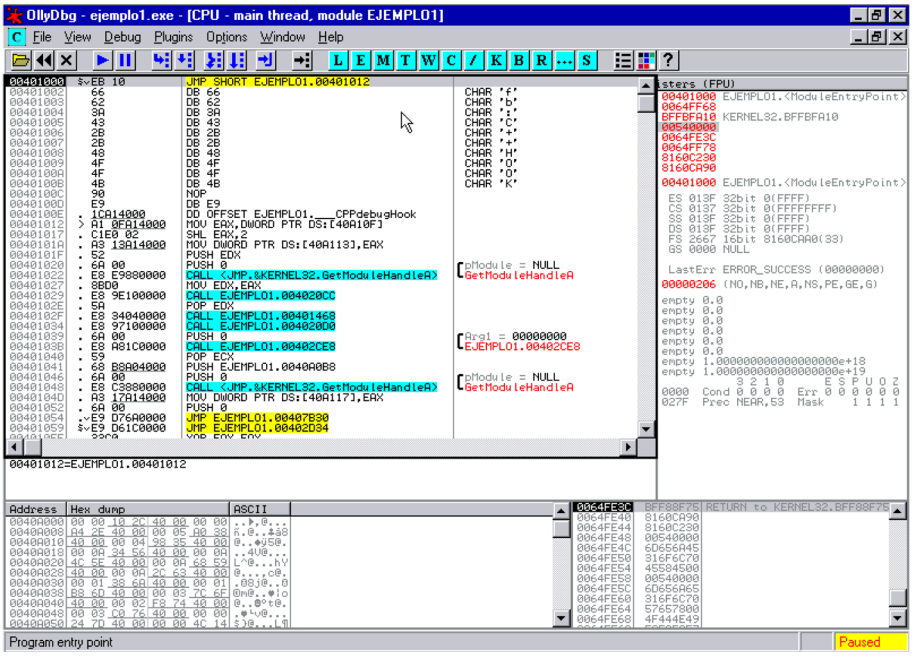
- Abrir archivo (F3).
- Reanudar el programa (Ctrl+F2).
- Cerrar el programa (Alt+F2).
- Ejecutar (F9).
- Pausa en la ejecución (F12).
- Ejecutar una instrucción (F7).
- Ejecutar una instrucción sin entrar en las llamadas (F8).
- Ejecutar automáticamente las instrucciones (animación) entrando en las llamadas (Ctrl+F11).
- Ejecutar automáticamente las instrucciones (animación) sin entrar en las llamadas (Ctrl+F12).
- Ejecutar hasta salida de función (Ctrl+F9).
- Ir a una instrucción en el *desensamblador*.
- Mostrar la ventana de Log (Alt+L).
- Mostrar la ventana de módulos cargados (Alt+E).
- Mostrar la ventana de memoria (Alt+M).
- Mostrar la ventana de *threads*.
- Mostrar la ventana de "ventanas".
- Mostrar la ventana de CPU (Alt+C).
- Mostrar la ventana de *patches* (Ctrl+P).
- Mostrar la ventana de llamadas (Alt+K).
- Mostrar la ventana de *breakpoints* (Alt+B).
- Mostrar la ventana de referencias.
- Mostrar la ventana de ejecución de un *trace*.

- Mostrar la ventana del código fuente.
- Configuración de opciones de *debug* (Alt+O).
- Opciones de apariencia.
- Ayuda.

Se puede ver que la cantidad de opciones es muy grande. En cada caso se utilizará tan solo un subconjunto de todas las herramientas disponibles.

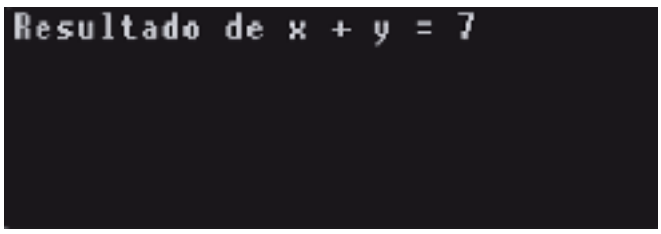
1.2.3. Modificar un programa en ejecución

Se procede a continuación a alterar el programa `ejemplo1` en tiempo de ejecución, de la misma manera que se ha hecho anteriormente, para que se muestre un valor distinto (123) por pantalla al final de la ejecución. La primera acción a tomar es cargar el programa en memoria para poder proceder a su análisis.



OllyDbg - Ejemplo1 cargado

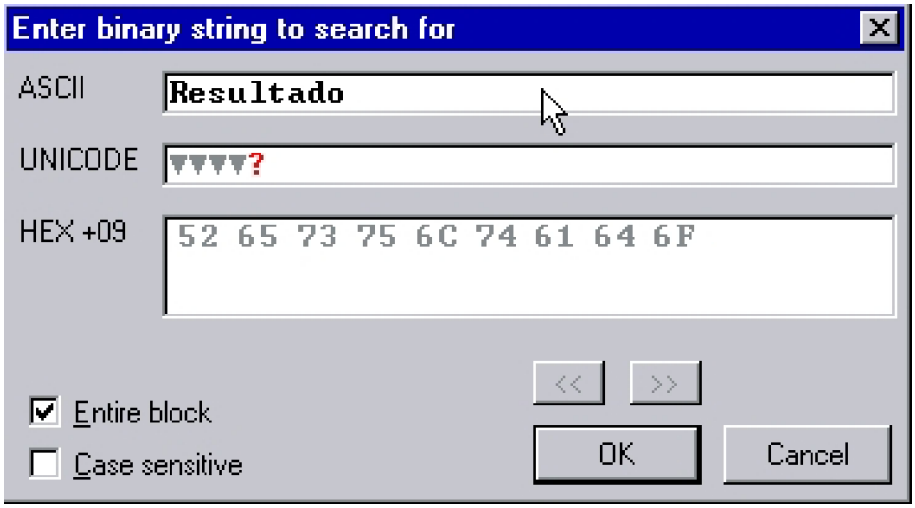
El siguiente paso consiste en localizar el punto en el cual se quiere actuar. Del programa ejecutado se conoce la salida que da tras su ejecución:



OllyDbg - Resultado ejemplo1

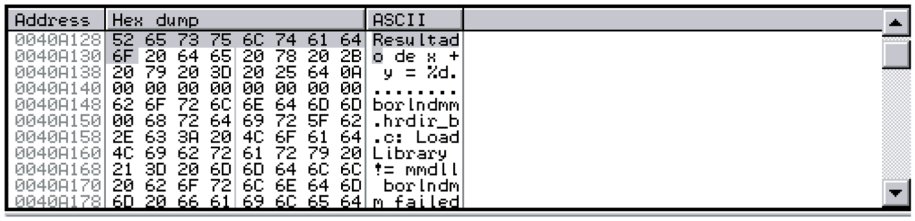
La cadena de caracteres `Resultado de...` es devuelta por el programa como salida. Esta información proporciona una idea del punto en el cual se puede actuar para modificar el resultado. Una opción posible es buscar la cadena de caracteres en la memoria del programa. Se procede a hacer clic con el bo-

tón derecho del ratón sobre la sección de memoria, seleccionando la opción "*Search for*" y del submenú "*Binary string*" (se puede conseguir el mismo resultado con la combinación de teclas "Ctrl+B". Aparece una ventana en la que se puede especificar el texto a buscar).



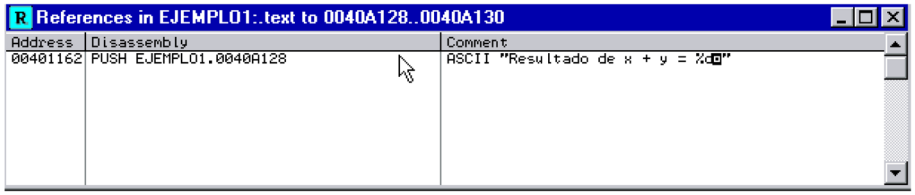
OllyDbg - Buscar texto

La ejecución de esta sentencia localiza en la memoria de la aplicación el texto. El resultado aparece en la sección de volcado de memoria.



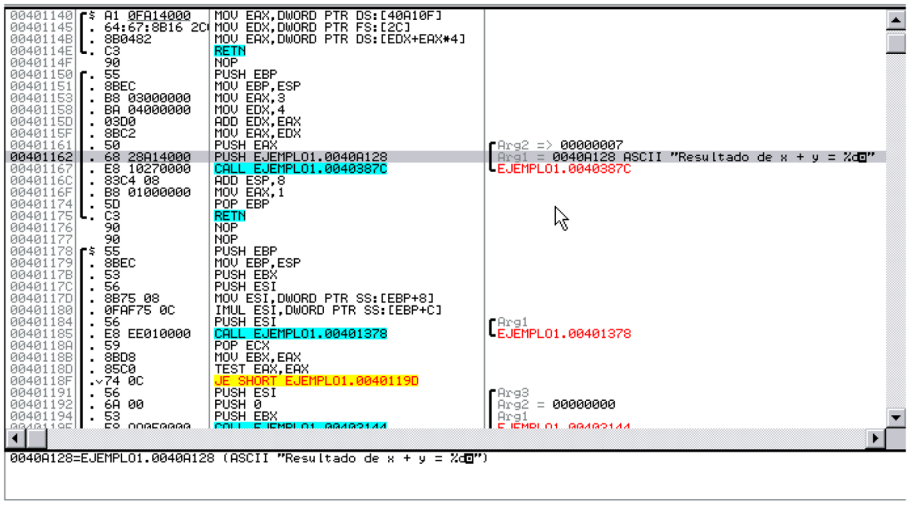
OllyDbg - Resultado de la búsqueda de texto en memoria

Ahora se necesita saber en qué punto de la aplicación se accede (o se hace referencia) a este dato. Muy probablemente, en alguno de esos puntos estará la sentencia que muestre por pantalla la información que se desea alterar. Para conseguirlo se hace clic con el botón derecho del ratón sobre el primer carácter de la cadena buscada y se selecciona la opción "*Find references*". También se puede presionar "Ctrl+R".



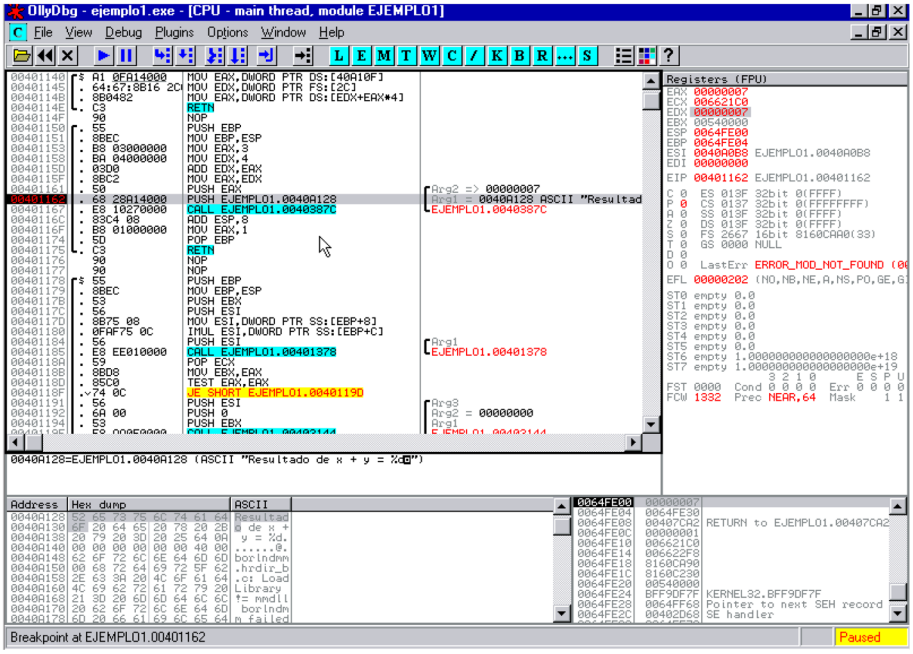
OllyGbd - Ventana de referencias

El resultado muestra una línea de código que introduce en la pila una referencia a la cadena. Haciendo clic con el botón derecho del ratón se puede elegir la opción "*Follow in disassembler*" para abrir la ventana de "CPU" y mostrar la línea de código de interés.



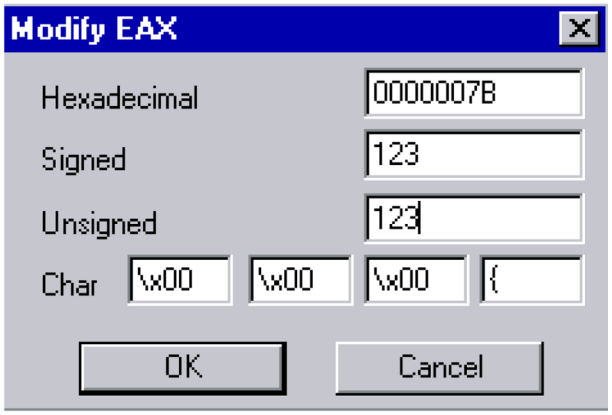
OllyDbg - Código que hace referencia la cadena buscada

Se puede observar que hay una llamada a una función (de la que no se conocen más datos que su dirección), a la que se le pasan dos parámetros en la pila. Un entero y una cadena de caracteres, que tiene un formato muy parecido al que usa la instrucción `printf` para mostrar texto por pantalla. La cadena de caracteres contiene el texto `%d` que en `printf` se interpreta como el comodín para mostrar un número entero. Este número podría ser el segundo parámetro. Teniendo esto en cuenta, se procede a marcar la línea con un *breakpoint* y a ejecutar el programa. Un *breakpoint* se puede crear haciendo doble clic en el código máquina de la línea deseada. La ejecución se consigue presionando F9.



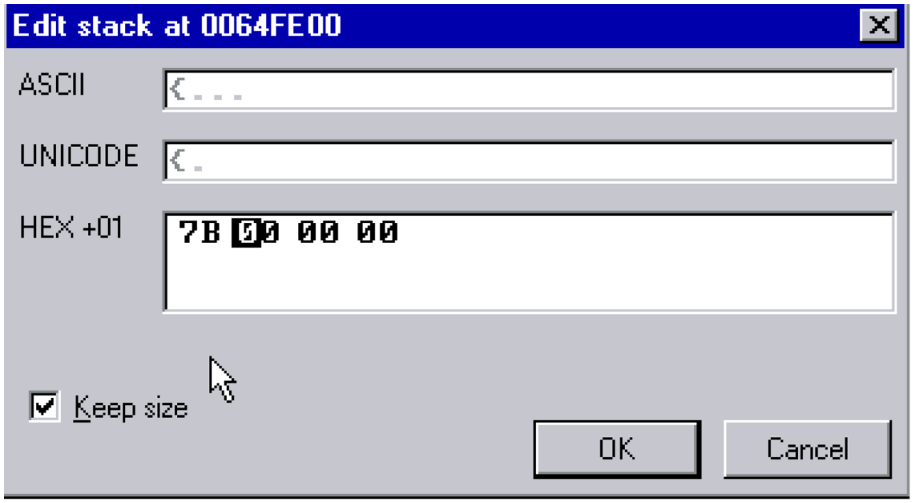
OllyDbg - Breakpoint antes de la llamada a una supuesta función `printf`

En la imagen se observa cómo se ha detenido la ejecución en el punto seleccionado. El valor de EAX es 7, cosa que concuerda con la salida del programa. Para alterar su ejecución hay que cambiar su valor al nuevo valor deseado. Como ya se ha introducido este valor en la pila, deberá cambiarse también allí. Si se hubiera puesto el *breakpoint* antes de introducir el valor de EAX en la pila, el trabajo de cambiar el valor de la pila no habría sido necesario. Se procede a cambiar el valor del registro EAX haciendo doble clic sobre él:



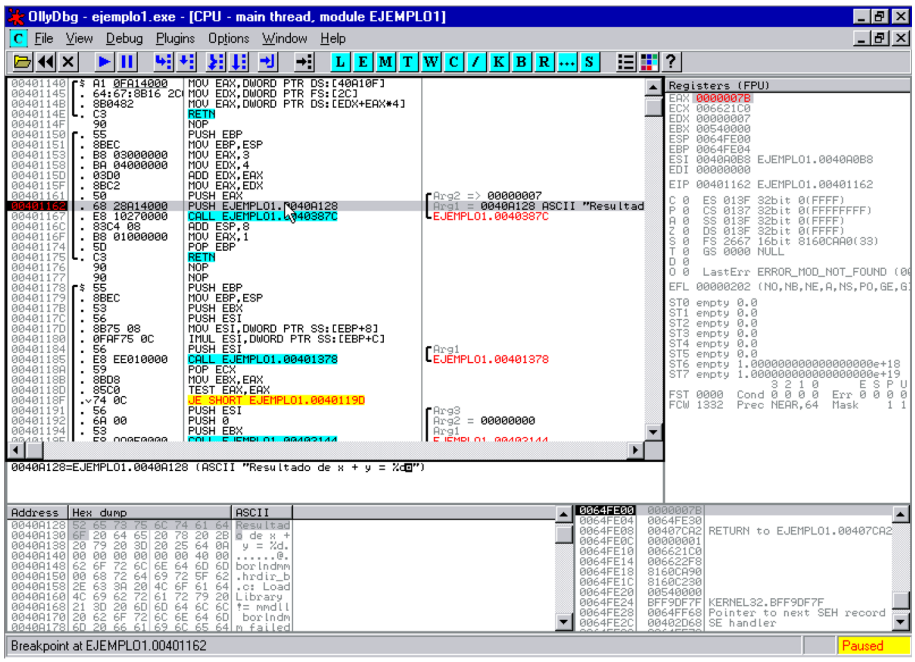
OllyDbg - Cambiar EAX

En la figura se le asigna el nuevo valor `123`. Se procede ahora a cambiar el valor introducido en la pila, como resultado de haber ejecutado un *push* del registro EAX. Se hace clic con el botón derecho del ratón sobre el dato de la pila que se quiere cambiar y se selecciona la opción `Edit` (Ctrl+E).



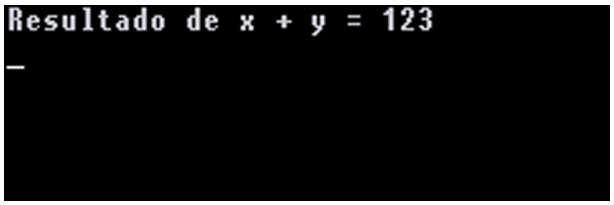
OllyDbg - Editar un dato de la pila

En este caso es necesario hacer el cambio en hexadecimal. Se cambia el valor y se guarda. El resultado mostraría la siguiente información en pantalla:



OllyDbg - Resultado modificado

Una vez que ya están acabados todos los detalles del cambio que se quiere realizar, se puede proceder a la ejecución del código y ver si así se ha obtenido el resultado deseado. Se presiona dos veces F8 (*step over*) para ejecutar la función que "probablemente" muestre la información de interés por pantalla.



OllyDbg - Resultado final

Como se puede observar, se ha conseguido el resultado deseado, porque ahora el programa no ha mostrado el 7 esperado, sino el `123` deseado.

1.2.4. Conclusiones

Este *debugger* es muy visual y ofrece un entorno bastante bueno para introducirse en el mundo del código máquina y ensamblador. Se dispone de mucha información accesible tan solo mirando a otras partes de la ventana activa. Evidentemente, no se han visto todas las opciones disponibles en esta aplicación, puesto que el interés de este módulo es introducir la herramienta para poder utilizarla en otros casos que se proponen en módulos posteriores. La herramienta realiza la inspección de un sistema a bajo nivel durante la ejecución de un programa, ofreciendo un entorno perfectamente usable para la realización de *exploits*.

2. Compiladores/lenguajes

Existen multitud de compiladores y lenguajes de programación en el mercado. Todos tienen sus ventajas e inconvenientes. De entre todos ellos, los más utilizados para la generación de *exploits* y búsqueda de vulnerabilidades son tres.

- Ensamblador.
- C.
- Perl.

Como se ha visto, el lenguaje ensamblador se usa para el análisis de aplicaciones a bajo nivel durante la ejecución de estas. Es el lenguaje que más cerca está de las instrucciones que ejecuta la máquina (código máquina). Esto ofrece un muchas posibilidades y opciones a la hora de buscar o explotar vulnerabilidades en programas.

El lenguaje de programación C se usa habitualmente por su flexibilidad a la hora de programar determinadas aplicaciones y por el elevado rendimiento de los programas realizados en dicho lenguaje. Se usa con mucha frecuencia para programar *exploits*.

En algunos casos también se usa Perl para la realización de *exploits*, aunque lo más habitual es usar lenguaje C. Normalmente el uso de *exploits* está asociado a entornos tipo Unix, aunque no tiene por qué estar limitado a ellos.

Teniendo en cuenta todo lo visto hasta el momento, resulta interesante ver cómo se puede obtener código máquina a partir de una serie de sentencias en ensamblador. De esta manera se podrían modificar apropiadamente las instrucciones a ejecutar por el procesador en un programa (cambiar el código durante su ejecución mediante un compilador) o crear *shellcodes* a medida. La creación de *shellcodes* se ve en otro módulo del curso, pero es interesante conocer cómo se puede obtener el código máquina a partir de ensamblador.

Modificar una aplicación durante su ejecución y hacer cambios permanentes en el fichero ejecutable para que la aplicación realice una serie de cambios introducidos por el usuario es lo que se conoce como *cracking*. Las herramientas vistas hasta el momento se pueden utilizar con este propósito, aunque no es el objetivo de este curso.

2.1. Generar "*opcodes*"

Para obtener el código máquina a partir de una serie de sentencias escritas en dicho código se puede crear un programa en ensamblador y compilarlo. Existen varios ensambladores en el mercado como "nasm[4]" o "as[4]".

(4)**NASM**. Disponible en: http://www.nasm.us/

Se considera el siguiente programa en ensamblador:

```
data # start of data segment

text # start of code segment

globl _start
_start:
   movl $4, %eax
   xorl %ebx, %ebx
   call *0x80431200
   ret
```

El programa no realiza ninguna operación concreta, pero ilustra el caso de necesitar el código máquina correspondiente a una serie de operaciones en ese código. Se puede ensamblar el programa anterior con "as":

```
as ensamblador.s
```

El resultado, desensamblado el código máquina anterior con *objdump*, es:

```
b8 04 00 00 00       mov $0x4,%eax
31 db                xor %ebx,%ebx
ff 15 00 12 43 80    call *0x80431200
c3                   ret
```

Utilizando el gdb se pueden obtener los códigos hexadecimales (*opcodes*) de manera casi inmediata para su uso en un *shellcode*.

```
(gdb) x /100xb 0x0
0x0 <_start>:     0xb8 0x04 0x00 0x00 0x00 0x31 0xdb 0xff
0x8 <_start+8>:   0x15 0x00 0x12 0x43 0x80 0xc3 Cannot access mem...
```