Eines

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

PID_00208380

UOC
Universitat Oberta de Catalunya

Índex

Introducció

Per a poder analitzar vulnerabilitats o *exploits* es necessiten coneixements bàsics de diverses eines. En primer lloc es troben els depuradors. Aquestes aplicacions permeten analitzar, a baix nivell (codi màquina i assemblador), l'execució de les aplicacions. Així és possible detectar vulnerabilitats i planejar l'escriptura de programes que alterin l'execució normal d'altres programes.

En segon lloc es troben els llenguatges de programació. Molts dels *exploits* que hi ha estan escrits en el llenguatge de programació C. D'altra banda, és molt necessari conèixer conceptes bàsics de llenguatge assemblador per a poder analitzar aplicacions a molt baix nivell, cosa que proporciona un gran control sobre l'execució dels programes.

1. Depuradors

Hi ha molts depuradors al mercat. N'hi ha de molt estesos, i d'altres que són populars. Amb Linux el depurador més potent és el GDB[1] (*The GNU Project Debugger*). També es pot executar en els sistemes Unix més populars i Microsoft Windows.

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

En plataformes Windows hi ha un ventall d'opcions quant a depuradors, encara que per a introduir-se en el món de la depuració en assemblador, tal vegada el més recomanat és l'OllyDbg[2]. Una vegada adquirits els coneixements bàsics amb l'OllyDbg, es poden utilitzar fàcilment amb altres eines com el WinDbg[2].

(2)**OllyDbg**. Disponible a http://www.ollydbg.de/

En aquest mòdul veurem com podem usar aquestes eines per a controlar l'execució de processos a baix nivell (codi màquina i llenguatge assemblador). En els exemples que es mostren no s'executen *exploits*, sinó que es trien petites aplicacions per a ser modificades en temps d'execució directament des del depurador. Molts *exploits* modifiquen les aplicacions que ataquen en temps d'execució per a aconseguir executar el seu propi codi. Els exemples no són *exploits* en si mateixos, però introdueixen el concepte de canviar el comportament de les aplicacions durant l'execució d'aquestes. El concepte és usat en molts *exploits* que, utilitzant vulnerabilitats de les aplicacions, canvien el seu comportament perquè facin el que l'atacant vol.

1.1. GDB

Aquest és el depurador usat en plataformes GNU/Linux. Té una interfície d'ordres molt potent que permet fer un conjunt molt extens d'operacions. La manera de treball normal del GDB és amb l'intèrpret d'ordres. Hi ha diverses aplicacions que, comunicant-se directament amb el GDB, afegeixen una interfície gràfica al depurador per a fer-lo més manejable. Encara que aquestes interfícies són molt pràctiques, la potència de GDB es troba en l'intèrpret d'ordres.

Presentarem aquí les operacions bàsiques per a poder traçar l'execució d'aplicacions amb aquesta eina. Hi ha molts llenguatges d'alt nivell que funcionen amb aquesta eina, i permet el seguiment a alt nivell de les aplicacions si el codi font és disponible. Així i tot, tant si el codi font està disponible com si no, el GDB pot fer el seguiment de l'aplicació a baix nivell (assemblador).

1.1.1. Primer contacte

L'entorn del depurador es pot invocar amb:

```
$ gdb
```

D'aquesta manera, s'entra en el mode interactiu d'ordres sense cap programa carregat. En la crida del depurador es poden especificar ordres.

```
$ gdb programa
```

D'aquesta manera, es carrega el programa especificat, i es prepara tot l'entorn per a l'execució. L'execució del programa en si no començarà fins que no s'especifiqui apropiadament en la línia d'ordres. Hi ha dues maneres més de cridar el depurador. La primera és:

```
$ gdb programa core
```

Aquesta és la manera de cridar el GDB per a traçar un estat d'excepció d'un programa que ha acabat normalment. El fitxer *core* conté l'estat de finalització del programa. D'aquesta manera, es pot inspeccionar, a baix nivell, què és el que va causar la finalització inesperada del programa. Per a generar els fitxers *core* el sistema ha d'estar configurat per a això. Aquest sistema és molt usat en desenvolupaments de programari.

L'últim mode de crida és:

```
$ gdb programa pid
```

Aquesta sentència llança el depurador i l'associa a l'execució d'un procés ja iniciat. El procés s'especifica amb el paràmetre `pid`. Una vegada feta la crida al depurador, el procés es deté i el GDB en pren el control.

Quan s'ha invocat l'intèrpret d'ordres del GDB apareix el *prompt*:

```
(gdb)
```

A partir d'aquest moment es poden executar ordres.

1.1.2. Ordres bàsiques

Aquest depurador té una ajuda en línia molt completa. L'ordre `help` ofereix en tot moment informació sobre les ordres disponibles. Executant `help` sense cap paràmetre s'obtindrà una ajuda general, on es mostren les diferents seccions en les quals estan dividides les ordres.

```
(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)
```

Escrivint `help` seguit de la secció en la qual s'està interessat, es proporcionarà una llista de les ordres disponibles en aquesta secció. Es pot tenir informació de cada ordre escrivint l'ordre `help` més la instrucció:

```
(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)
```

L'entorn ofereix la capacitat d'autocompletar les ordres mitjançant l'ús de la tecla *TAB*. Així, doncs, pressionant aquesta tecla mentre s'està escrivint una ordre, el sistema la intentarà completar. Si no és capaç de fer-ho, oferirà una llista d'opcions, perquè l'usuari pugui triar.

- `run`: execució del programa. Aquesta ordre executa el programa carregat en memòria. També es pot executar amb `r` solament. Se li poden passar paràmetres, com es passarien al programa que s'està analitzant, des de l'intèrpret d'ordres.

- `start`: execució del programa i aturada. Aquesta ordre executa l'ordre carregada en memòria i en deté l'execució al principi de l'execució del codi principal del programa. En cas de tractar-se d'un programa escrit en C, es detindria al principi de la funció `main`. Això només és vàlid si l'executable disposa de la taula de símbols. En cas contrari l'execució del programa continuarà fins al final, i retornarà missatges que diuen que no es troba la taula de símbols.

- `continue`: represa d'execució. Aquesta ordre reprèn l'execució del programa després que hagi estat detinguda amb algun dels mètodes que permet el GDB. També es pot executar amb `c` solament.

- `step`: executa una instrucció del programa. Executa una instrucció del programa, i entra en les crides a funcions si és necessari. Aquesta ordre està associada a l'execució d'una instrucció escrita en un llenguatge d'alt nivell. Si tan sols es disposa de codi assemblador, no s'ha d'utilitzar, ja que el resultat pot ser impredictible. També es pot executar aquesta ordre amb `s`.

- `stepi`: executa una instrucció del programa (*asm*). Executa una instrucció del programa a baix nivell, i entra en les crides a funcions si és necessari. Aquesta ordre està associada a l'execució d'instruccions en llenguatge màquina. És l'ordre recomanada si no es disposa del codi font del programa. També es pot executar aquesta ordre amb `si`.

- `next`: executa una instrucció del programa. Executa una instrucció del programa sense entrar en les crides a funcions. Aquesta ordre està associada a l'execució d'una instrucció escrita en un llenguatge d'alt nivell. Si tan sols es disposa de codi assemblador, no s'ha d'utilitzar, ja que el resultat pot ser impredictible. També es pot executar aquesta ordre amb `n`.

- `nexti`. Executa una instrucció del programa a baix nivell, i entra en les crides a funcions si és necessari. Aquesta ordre està associada a l'execució d'instruccions en llenguatge màquina. És l'ordre recomanada si no es disposa del codi font del programa. També es pot executar aquesta ordre amb `ni`.

- `break`: definició de punts de ruptura. Una de les instruccions més interessants és `break`. S'utilitza per a definir punts (punts de ruptura) en la línia d'execució del programa on s'interromp l'execució, per a donar el control del procés a l'usuari del GDB. Una vegada es té el control es poden veure/canviar dades de memòria i registres i després reprendre l'execució normal del programa fins a un altre punt de ruptura o fins a la finalització. També es poden executar instruccions pas per pas, per a poder veure com evoluciona l'execució del programa.

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

En l'exemple s'ha definit un punt de ruptura en una adreça de memòria especificada. Les adreces de memòria han d'anar precedides d'un asterisc (`*`). Si el programa que s'executa té la seva taula de símbols, llavors es poden especificar etiquetes (com noms de funcions) per a definir els diferents punts de *break*. Si el programa disposa d'informació de depuració, llavors fins i tot es té accés al codi font des del depurador. Aquesta situació és bastant improbable en programes en sistemes de producció.

- `info break`: informació sobre els punts de ruptura definits. Aquesta ordre retorna una llista amb els *breakpoints* definits fins llavors.

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

- `tbreak`: punt de ruptura temporal. Aquesta ordre fa la mateixa funció que `break`, però de manera temporal. De tal manera que quan el punt de ruptura ha estat utilitzat una vegada, es desactiva automàticament.

```
(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 la llista apareixen dos punts de ruptura, un que és permanent (`keep`) i un altre que és temporal (`del`). `Keep` indica que una vegada s'hagi aconseguit el punt de ruptura, aquest s'ha de conservar per a usos posteriors. `Del` indica que quan s'aconsegueixi el punt de ruptura, aquest sigui esborrat.

- `watch`: definició de *watchpoints*. Deté l'execució del programa (igual que els punts de ruptura) quan el valor apuntat per l'expressió definida pel paràmetre canvia (escriptura de valors). En aquest moment, el control del programa és retornat a l'usuari i li mostra el valor antic i el valor nou:

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

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

Hi ha un parell de variants de *watch:*

- `rwatch`: deté l'execució del programa si el valor apuntat és llegit.

- `awatch`: deté l'execució del programa si el valor apuntat és llegit o escrit.

- `delete`: esborra un punt de ruptura. Elimina un punt de ruptura de manera definitiva. Cal indicar el nombre de punts de ruptura per eliminar, tenint en compte la llista proveïda per `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 temporalment un punt de ruptura. Al contrari que la instrucció *delete*, l'activació/desactivació dels punts de ruptura és temporal. Es pot observar l'estat d'aquests 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 la llista s'observa com l'estat (indicat per la columna `Enb`) del punt de ruptura número 2 canvia de `y` a `n`, i després de `n` a `y`. La columna `Enb` indica si el punt de ruptura està `enabled`.

- `ignore`: ignora un nombre determinat de passades sobre un punt de ruptura.

```
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 l'exemple es defineix el punt de ruptura número 2 i s'indica que ha de ser ignorat una vegada. En el moment de fer això, el programa està detingut en el punt de ruptura número 1. Després de fer la definició del punt de ruptura número 2, es reprèn l'execució del programa. Aquest acaba a causa que el programa en execució no passa dues vegades pel punt de ruptura número 2.

Aquest tipus de punt de ruptura és útil per a avançar, de manera controlada, en l'execució de bucles.

- `finish`: continua l'execució fins a la finalització de la funció actual.

Si el depurador es troba en una funció que pot reconèixer i s'executa aquesta instrucció, l'execució del programa continua fins que la funció finalitza, i retorna l'execució a la línia següent de la crida a la funció.

- `display`: activa la visualització constant d'un valor.

Aquesta ordre crea una visualització continuada d'un valor especificat. Cada vegada que el programa es deté es mostra el valor indicat.

```
(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 l'exemple es defineix un `display` que mostra en tot moment la posició de memòria `0xbfceea80`. L'adreça de memòria s'especifica com a segon paràmetre de l'ordre. La primera ordre especifica el format amb el qual es mostraran

les dades. La definició del format ha de començar amb una `/`, a continuació el nombre d'elements per mostrar, després el tipus de valor per mostrar i finalment la mida de cada element mostrat. Els possibles tipus de valors són:

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

La mida dels elements pot ser:

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

Així, doncs, en l'exemple s'especifica que s'ha de mostrar una dada en hexadecimal de mida *word*.

- `info display`: mostra informació sobre *displays*. Aquesta ordre mostra els *displays* definits.

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

En l'exemple es mostra que hi ha un `display` definit i activat. Els *displays* es poden activar i desactivar utilitzant les ordres `enable/disable display`. El funcionament és igual al descrit per a l'ordre `enable/disable` utilitzada en els punts de ruptura.

- `undisplay`: esborra un *display* definit. Aquesta ordre elimina de la llista de *displays* definits l'element seleccionat. L'ordre següent esborra el *display* definit en l'exemple anterior.

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

- `x`: examinar. Aquesta ordre és molt semblant a l'ordre `display` quant a la sintaxi. Consta també de dos paràmetres. El primer és el format i el segon és el valor per mostrar. En aquest cas es mostren les posicions de memòria especificades i no hi ha reiteració automàtica. Aquesta ordre s'usa per a explorar posicions de memòria.

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

- `print`: mostra una dada especificada. Aquesta ordre s'utilitza per a mostrar dades concretes de manera puntual. L'ordre `x` és molt útil per a explorar

la memòria, però per a mostrar una dada concreta és més pràctic utilitzar aquesta ordre.

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

El resultat de l'execució d'aquesta ordre és la informació demanada més una referència a aquesta. En l'exemple s'observa un `$4` afegit al principi de la resposta de l'ordre. Aquesta referència pot ser utilitzada en altres ordres, per a usar el valor que s'ha obtingut en ordres `print` anteriors.

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

- `info registers`: mostra els registres del processador.

```
(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
```

També es pot accedir al valor dels registres utilitzant alguna de les ordres anteriors de mostrar informació. Es pot fer referència als registres utilitzant el seu nom genèric i situant un `$` al principi. Observem els exemples següents:

```
(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 instrucció es mostra el contingut del registre EIP.

El segon exemple fa un buidatge de memòria dels 16 bytes que apareixen a continuació del punt on apunta el registre EIP.

El tercer exemple fa un buidatge del contingut de la pila (usant elements de mida *word*) a partir de l'adreça apuntada pel registre ESP.

- `info all-registers`: mostra tots els registres. L'ordre anterior mostra els registres generals, però el processador té més registres. Es pot veure la llista completa de registres usant aquesta ordre.

- `disassemble`: desassembla el codi màquina. Aquesta ordre desassembla el codi màquina en execució. Executada sense paràmetres desassembla la funció que es troba en execució en aquest moment i és capaç d'identificar. Perquè la pugui identificar és necessari que el programa contingui la taula de símbols. En cas que aquesta no sigui present, se li poden indicar un parell d'adreces de memòria perquè la desassembli.

```
(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)
```

Aquest exemple disposa de la taula de símbols i l'execució es trobava en un punt identificat per un element d'aquesta, amb la qual cosa el GBD ha pogut identificar la funció i l'ha desassemblat sencera.

```
(no debugging symbols found)
(gdb) break *0x080482f0
Breakpoint 1 at 0x80482f0
(gdb) r
Starting program: exemple1_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 aquest cas, el programa disposa de la taula de símbols i, per tant, el depurador no pot identificar funcions automàticament (com per exemple la funció `main`). Així, doncs, cal seguir l'execució del programa sense parar gaire atenció a la identificació de funcions que el depurador intenta fer. Una de les dades importants que cal conèixer per a poder executar aquest programa pas per pas és l'`entry point` o `start address` del programa. Per a aquesta comesa s'usa l'ordre `info target`.

- `info target`: dóna informació sobre el programa carregat. Una de les informacions més rellevants és l'`entry point` o `start address`.

```
(gdb) info target
Symbols from "exemple1_strip".
Local exec file:
exemple1_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 contingui la taula de símbols, aquesta informació és important per a poder començar la depuració del programa. El mètode usat habitualment és definir un punt de ruptura en l'adreça de l'`entry point` i començar la sessió de depuració a partir d'aquest punt.

- `set`: canvia el valor d'un element. Amb aquesta ordre es pot canviar el valor d'un element donat. Qualsevol posició de memòria o registre pot ser canviat 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 aquest exemple es canvia una posició de memòria situada en la pila, utilitzant com a referència el registre ESP i desplaçant l'objectiu 4 bytes dins de la pila. Per a poder canviar el valor s'ha hagut de fer un *typecasting* del valor desat en el registre ESP. Com que en la pila es poden desar valors de qualsevol tipus, el punter a la pila és un punter genèric. Per a poder escriure un valor és necessari proporcionar un tipus. El tipus que s'especifiqui marcarà la mida de les dades per escriure. En aquest cas s'ha escrit un 0 de mida `int` (4 bytes). Per a reescriure tan sols un byte s'haurà d'especificar el tipus `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)
```

Els registres també es poden alterar:

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

Fins i tot el codi del programa és alterable. Es pot canviar el codi màquina per a alterar-ne l'execució. Per a fer això, és necessari conèixer quins són els *opcodes* que es volen posar per a reemplaçar els existents. Abans de fer un canvi d'aquest tipus cal documentar-se sobre aquest tema, per a veure si el canvi que es vol fer és factible o no.

1.1.3. Execució d'un programa simple

Per a aquest exemple es considera un programa que quan s'executa sense paràmetres retorna el resultat següent:

```
$ exemple1
Resultat de x + y = 7
$
```

Aquest programa és l'exemple que ja hem tractat anteriorment. Es farà el traçat suposant dues situacions diferents:

- L'executable disposa de la taula de símbols.
- L'executable no té la taula de símbols.

Si un programa conserva la seva taula de símbols, fer-hi una sessió de depuració resulta molt més senzill.

1.1.4. Executable amb taula de símbols

Primerament es carrega el depurador amb el programa que es vol tractar:

```
$gdb exemple1
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 vegada a l'entorn d'execució, no es rep cap missatge que indiqui que el programa no té taula de símbols. Per tant, es pot intentar avançar fins a la funció principal del programa.

```
(gdb) start
Breakpoint 1 at 0x80483b2
Starting program: /home/jgay/projectes/uoc/bin/exemple1_nodebug
0x080483b2 in main ()
Current language: auto; currently asm
(gdb)
```

El depurador aconsegueix arrencar el programa i es té l'execució en la funció *main*, que detecta sense cap problema. Una vegada en aquesta funció, es pot desassemblar la funció automàticament:

```
(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)
```

Aquest és el codi assemblador de la funció *main*. Es pot observar la crida a la funció `printf`, i tenint en compte que el programa escriu una cadena de caràcters com a sortida, aquesta ha de ser la crida que fa aquesta operació. Just abans de la crida `call 0x80482d8 <printf@plt>` s'han de preparar els paràmetres en la pila perquè la crida a la funció tingui èxit. El primer paràmetre s'introdueix en la pila mitjançant la sentència prèvia a la instrucció `call`:

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

Amb la instrucció anterior s'introdueix en la part superior de la pila l'adreça de memòria indicada. S'examina llavors el contingut d'aquesta adreça de memòria:

```
(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: "Resultat de x + y = %d\n"
(gdb)
```

Es veu que l'adreça de memòria conté un punter a l'*string* que servirà de format per a imprimir el resultat per pantalla. D'aquesta cadena de caràcters es dedueix que falta un segon paràmetre, que serà un enter i indicarà el valor del resultat. El segon paràmetre s'introdueix en la pila amb la sentència anterior:

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

Atès que l'execució del programa encara no ha assolit aquest punt, veure quin és el valor d'aquest paràmetre donaria com a resultat un valor equivocat. S'avança, llavors, l'execució del programa fins a just abans de fer la crida a la funció `printf`, utilitzant la instrucció `ni (nexti)`.

```
(gdb) info program

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

En aquest punt ja es pot veure quin és el valor que es mostrarà per pantalla:

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

S'ha consultat el valor del registre EAX, però també es pot observar aquest valor en la pila, perquè ja està preparat per a ser passat com a paràmetre a la funció cridada.

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

A manera d'exemple, passem a alterar el contingut de la pila perquè el valor mostrat, quan s'executi la crida a `printf`, sigui un altre, com per exemple `123`:

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

Com que ha estat el contingut del registre EAX el que s'ha introduït en la pila, per a alterar el comportament del programa de manera coherent s'hauria de canviar també el contingut d'aquest registre al nou valor:

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

Una vegada fetes les alteracions que volem, es procedeix a avançar l'execució del programa executant la crida a la funció `printf`:

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

S'observa que el resultat del programa ha estat alterat correctament. A partir d'aquest punt s'executa el programa normalment fins a la finalització.

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

S'ha vist que l'execució d'un programa amb la seva taula de símbols és relativament senzilla, ja que no és necessari localitzar els diferents elements que s'executen, ja que el depurador és capaç de fer-ho per si sol.

1.1.5. Executable sense taula de símbols

Igual que en el cas anterior, es carrega el depurador amb el programa que es vol tractar:

```
$ gdb exemple1
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 aquest cas el depurador dóna un avís que no hi ha símbols disponibles; això significa que el GDB no pot localitzar automàticament les diferents seccions del programa i que, per tant, la localització de la funció principal *main* s'haurà de fer manualment. Es procedeix a fer la mateixa manipulació que en el cas anterior, o sigui que el programa retorni 123 en lloc de 7. Per a comprovar que el depurador no pot fer la detecció de la funció principal, executem la instrucció `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/exemple1_strip
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
Resultat de x + y = 7

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

En executar la instrucció, el depurador no troba el símbol demanat (`main`) i pregunta si pot ser que aquest símbol es trobi en alguna biblioteca que es carregui més endavant. Se li diu que sí perquè ho intenti, però el programa s'executa íntegrament sense detenir-se, ja que el símbol demanat no existeix. El pas següent consisteix a determinar l'`entry point` del programa i detenir l'execució en aquest punt.

```
(gdb) info target
Symbols from "exemple1".
Local exec file:
`exemple1', file type elf32-i386.
Entry point: 0x80482f0      0x08048114 - 0x08048127 is .interp
                            0x08048128 - 0x08048148 is .note.ABI-tag
                            0x08048148 - 0x08048170 is .hash
...
```

Una vegada trobada l'adreça d'entrada al programa, es defineix un punt de ruptura en aquesta adreça i s'executa 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/exemple1_strip
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)

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

Una vegada en aquest punt, es desassembla el codi que s'executarà. Cal tenir en compte que el depurador no reconeix cap funció en aquest punt (`Breakpoint 1, 0x080482f0 in ()`) i ho indica amb dos signes d'interrogació. Per a obtenir un desassemblatge del codi que s'executarà cal indicar l'interval de memòria que es vol desassemblar:

```
(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 possible manera de procedir seria col·locant un punt de ruptura en cadascuna de les adreces d'execució del programa, que s'introdueixen en la pila. Habitualment, l'última adreça introduïda en la pila és l'adreça de la funció principal *main*. Si aquesta informació no es considera interessant i es vol anar directament a fer la manipulació que volem, llavors es pot procedir d'una altra manera.

Com que es veu que el programa retorna un *string* amb un missatge, es poden veure quines són les funcions del sistema que aquest programa crida per a bolcar per pantalla la informació.

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

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

S'observa que el programa tan sols té 3 funcions definides en la secció PLT (*procedure linkage table*). Aquestes 3 funcions es criden des del programa. Així, doncs, la que interessa per a poder fer la manipulació desitjada és `printf`. L'adreça que s'especifica és el punt d'entrada a la crida a `printf`. Llavors es pot posar un punt de ruptura en aquesta adreça.

```
(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)
```

Es pot continuar amb l'execució del programa, i aquest es detindrà quan la funció `printf` sigui cridada. En aquest punt es podrà alterar, segurament, 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)
```

L'execució s'ha detingut just quan la crida a la funció `printf` ha estat feta. Això significa que l'últim element de la pila indica l'adreça de retorn de la crida, o sigui, una adreça que es troba en el fil d'execució principal del programa. També es coneix que els registres no han quedat alterats a excepció dels registres EIP (la instrucció per executar-se ha canviat) i ESP (s'ha apilat la instrucció de retorn).

```
(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 crida a la instrucció `disassemble` es fa utilitzant directament el registre ESP, ja que conté l'adreça de memòria d'un punter que al seu torn conté l'adreça de retorn de la crida a `printf`. Una altra manera de fer la crida seria especificant de manera explícita l'adreça en la qual es basa el desassemblatge.

```
(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 vegada s'arriba a aquest punt, es poden fer les mateixes modificacions al programa que el cas anterior, però tenint en compte que l'execució del programa es troba en un altre punt. Així, doncs, el paràmetre buscat es trobarà 4 bytes més endavant tenint en compte que s'ha apilat l'adreça de retorn.

```
(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)
```

S'han modificat els antics valors tant en la pila com en el registre EAX. Una vegada fet això, ja es pot procedir a l'execució del programa fins al final per a veure si el canvi ha estat correcte.

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

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

Tot acaba correctament, i mostra el nou resultat.

En aquests casos en els quals la taula de símbols ha estat esborrada és molt útil una eina que complementa el DGB i que es diu `objdump`. Aquesta eina permet mostrar una gran quantitat d'informació d'un fitxer executable. Entre les seves característiques podem trobar:

- Desassemblatge del programa.
- Mostra la taula de símbols.
- Mostra dades sobre les diferents seccions de l'executable.

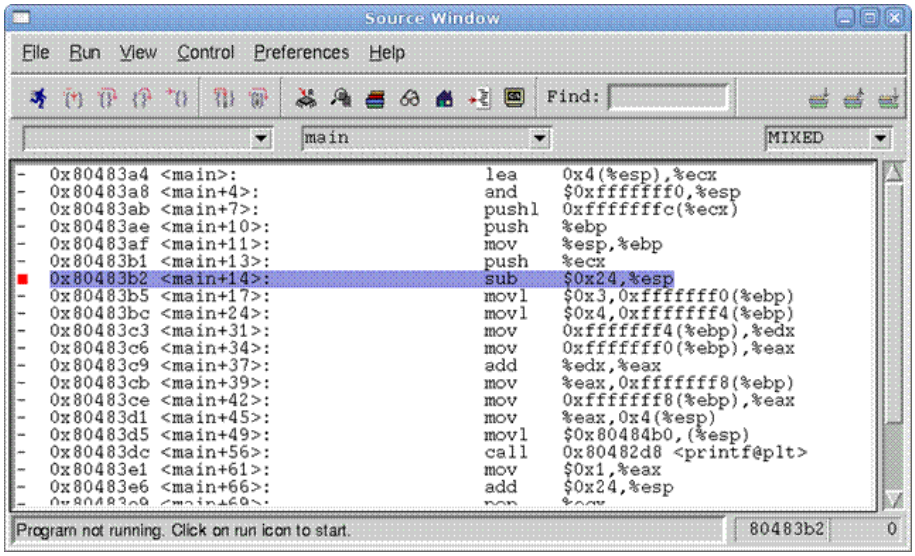
És molt útil per a fer una anàlisi separada del depurador. A causa que ofereix llistes d'una manera senzilla, és una eina molt usada per a analitzar codi.

1.1.6. *Insight*

El depurador GDB és una eina molt potent, però no és gaire intuïtiva. Per a millorar-ne la utilització, el depurador incorpora una petita interfície en mode text anomenada TUI (*text user interface*). Encara que és una opció interessant,

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

hi ha una altra possibilitat més còmoda. Hi ha una interfície gràfica per al GDB anomenada Insight[3]. Des d'aquesta interfície es té disponibilitat de tota la potència del GDB però d'una manera més còmoda. La vista general de l'Insight quan s'executa és la següent:



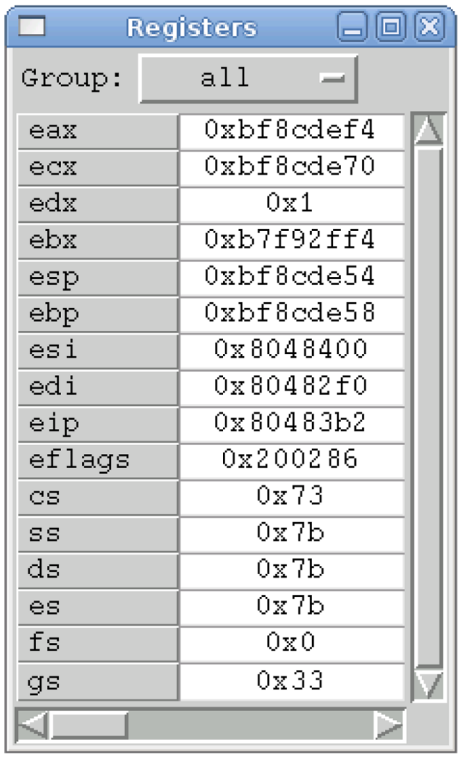
Insight: finestra del codi font

En aquesta finestra es pot observar el codi en assemblador del programa per executar i un punt de ruptura creat automàticament per l'entorn (això és configurable) en l'entrada de la funció principal del programa. Cal tenir en compte que el depurador només serà capaç d'oferir l'entorn en aquestes condicions si el programa que s'està tractant conté la taula de símbols. En cas que no sigui així no es mostrarà cap codi a l'entrada del programa. Tot i així, les diferents opcions del programa fan que sigui pràctic treballar-hi.

La finestra principal permet la inserció de punts de ruptura, tant fixos com temporals, usant el botó dret del ratolí.

L'entorn té una sèrie de finestres auxiliars que són d'ajuda durant el procés de depuració d'un programa. Entre les més interessants es troben:

- **Finestra de registres**. Aquesta finestra visualitza en tot moment els registres del processador.



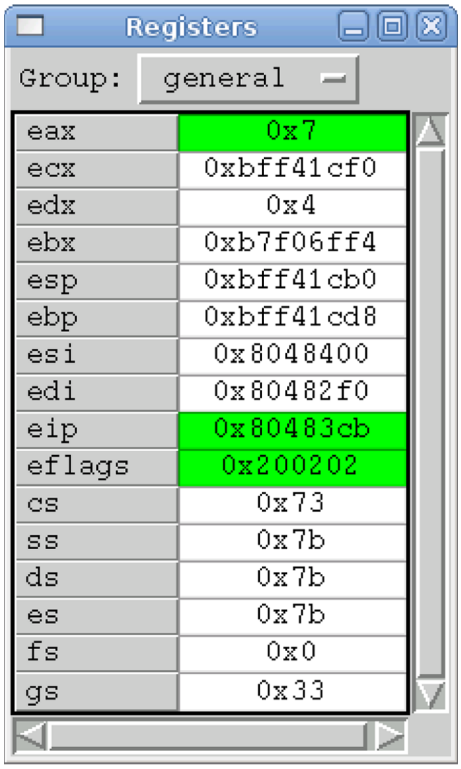
Insight: finestra de registres

Es pot escollir el subconjunt de registres visualitzats. Normalment el conjunt de registres més útil és el dels registres generals, però les possibilitats inclouen altres conjunts:

- `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 els registres de totes les diferents categories
- `vector`: agrupa els registres de la categoria `sse` i `mmx`
- `system: orig_eax`

Cadascun d'aquests conjunts de registres s'usa per a una tasca específica. Els registres en la categoria "general" són els més utilitzats i els que resulta més útil tenir controlats en tot moment.

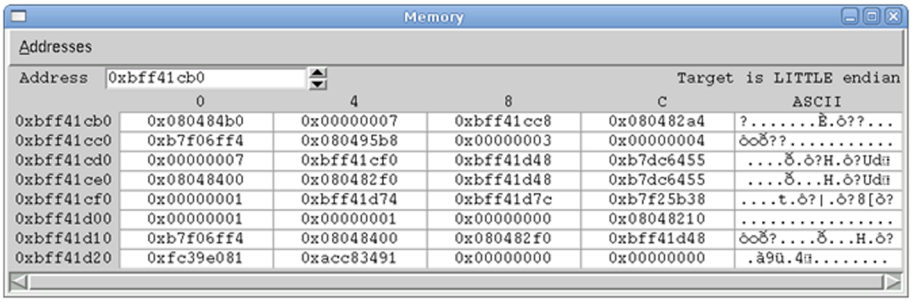
Una funció molt interessant d'aquesta finestra és que cada vegada que un dels registres visualitzats canvia de valor, el sistema el canvia de color per indicar que ha estat modificat. Això és molt interessant quan s'està executant un programa.



Insight: finestra de registres (canvis)

També es pot canviar el contingut d'un registre situant el cursor sobre la casella del registre corresponent i introduint-hi un valor nou.

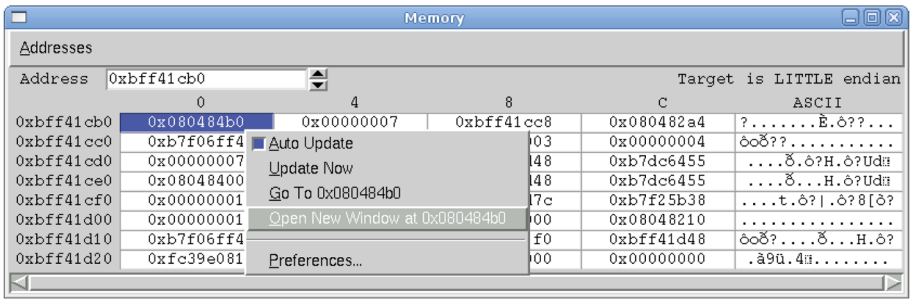
- **Finestra de buidatge de memòria**. En aquesta finestra s'inspecciona una zona de la memòria. El nombre i la mida dels elements per visualitzar és configurable.



Insight: finestra de memòria

Es poden obrir tantes finestres d'aquest tipus com es vulgui, per a explorar diferents posicions de memòria en cadascuna. Es pot pressionar el botó dret del ratolí sobre un registre (per exemple, `ESP`) i seleccionar l'opció "Open memory window". Apareixerà una finestra com l'anterior que mostrarà l'adreça seleccionada en la primera adreça de memòria del bloc.

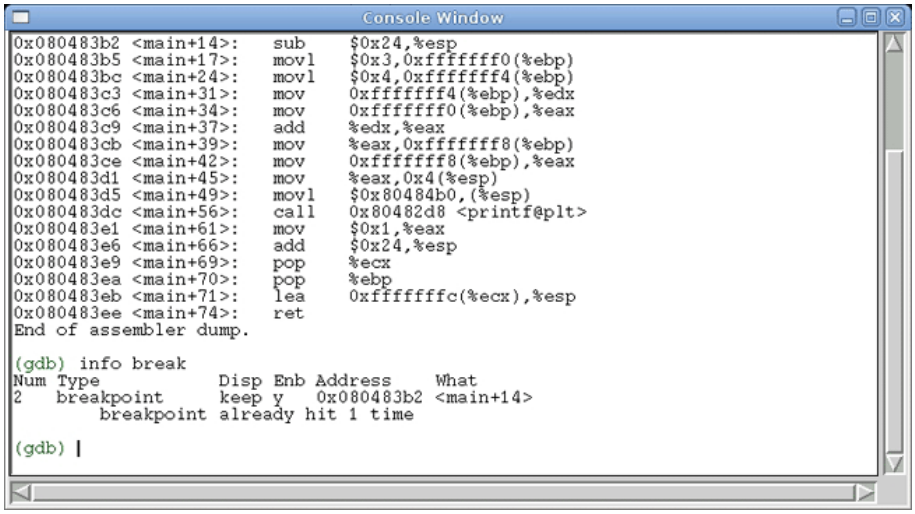
A més, cada posició de memòria és editable i es pot variar el valor de cada cel·la tan sols seleccionant la cel·la i canviant el valor. Pressionant el botó dret del ratolí sobre una cel·la, es pot seleccionar l'opció "Open new window at xxxx", i s'obrirà una nova finestra que explorarà una nova regió de memòria.



Insight: obrir nova finestra de memòria

El resultat serà una nova finestra que mostra el contingut de la nova posició de memòria triada.

- **Finestra de consola**. La finestra de consola ofereix l'intèrpret d'ordres de GDB, com ja hem vist anteriorment. Totes les ordres del depurador queden a la disposició de l'usuari, que disposa a més dels avantatges de l'entorn gràfic.

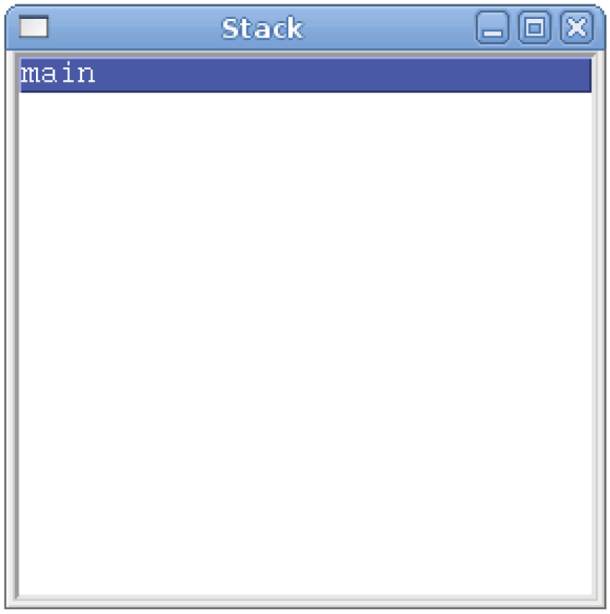


Insight: finestra de consola

En cas que no es disposi de la taula de símbols en l'executable, la finestra de consola es converteix en la finestra principal durant la sessió de depuració. Les finestres de registres i de memòria s'usen per a fer un seguiment de l'estat del programa en execució. Si es disposa de la taula de símbols, llavors la finestra del codi font es torna més útil.

- **Finestra amb la llista de les funcions cridades (*frame stack*)**. Aquesta finestra resulta útil si es disposa de la taula de símbols; si no, no és d'utilitat. S'hi mostra la seqüència de crides entre funcions. Es pot canviar d'una

funció a una altra en la llista, i es pot consultar informació de l'estat de cadascuna de les crides.



Insight: llista de funcions cridades

Encara que l'Insight és una eina interessant, cal no descuidar que el motor de tot aquest procés és el depurador GDB. L'entorn gràfic no deixa de ser un afegit a una eina que ja per si mateixa és totalment útil.

1.1.7. Conclusions

Aquest depurador és una eina molt potent. Permet analitzar a baix nivell tot tipus d'estructures executades, consultades i modificades per un programa durant l'execució. Encara que no hem vist totes les opcions possibles, hem donat una idea bàsica del seu funcionament per a ajudar a depurar programes. Aquesta depuració a baix nivell és molt útil quan s'estan escrivint *exploits*, tant per a analitzar el punt en el qual és (o pot ser) vulnerable un programa com per a veure si l'*exploit* que s'està escrivint fa correctament els canvis que es volen en el programa víctima. També en el procés d'aprenentatge de la utilització d'aquesta eina s'entra en contacte amb elements del sistema que normalment passen desapercebuts per a un usuari normal.

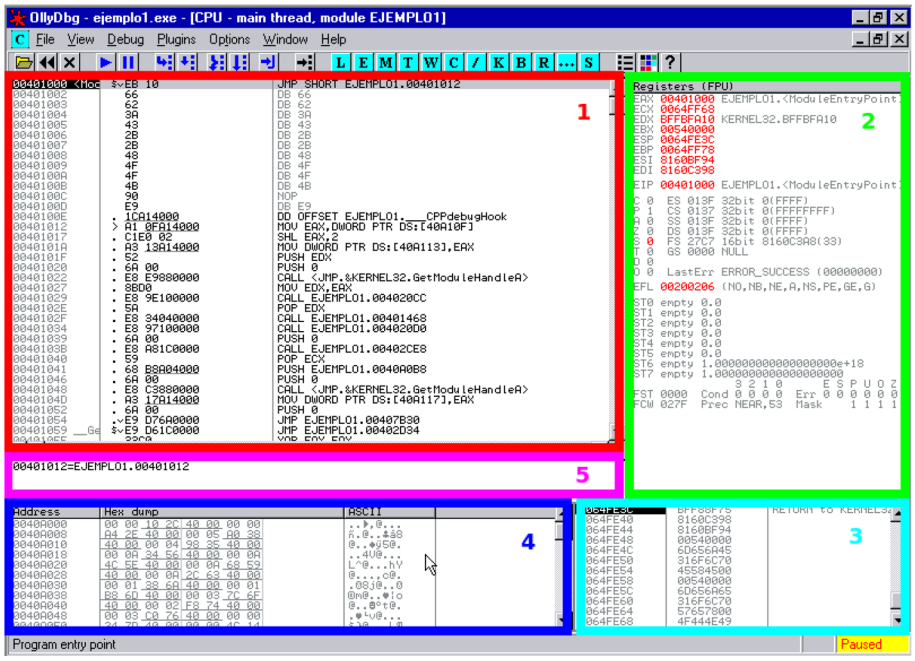
1.2. OllyDbg

L'OllyDbg és un depurador molt utilitzat en entorns Windows, encara que no és l'únic. Microsoft té la seva pròpia eina, que és el WinDbg, i fins i tot el GDB té una versió per als sistemes operatius de Microsoft. Cal tenir present que els conceptes per a seguir l'execució de programes a baix nivell no canvien, ja que tan sols s'està canviant el programari, no el maquinari. Per tant, es continuen tenint els mateixos registres, la inspecció de memòria continua essent un punt

interessant i el desassemblatge de les diferents instruccions de codi màquina serà equivalent. L'única manera de fer que hi hagués un canvi gran en la manera de fer l'anàlisi seria que es tingués un canvi d'arquitectura maquinari.

1.2.1. Primer contacte

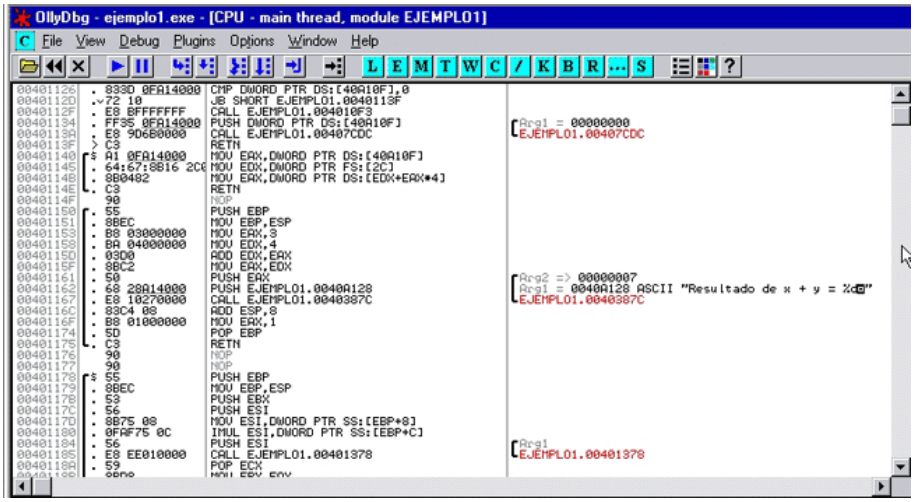
L'OllyDbg és una aplicació que té una finestra principal en la qual es van obrint altres subfinestres que donen accés a diferents opcions o funcionalitats de l'aplicació. Si s'obre el depurador sense carregar cap aplicació, tan sols s'obtindrà una finestra buida, amb el menú i unes icones per a poder utilitzar. Si en aquest moment s'obre una aplicació apareixerà una finestra com la de l'exemple:



OllyDbg: finestra principal

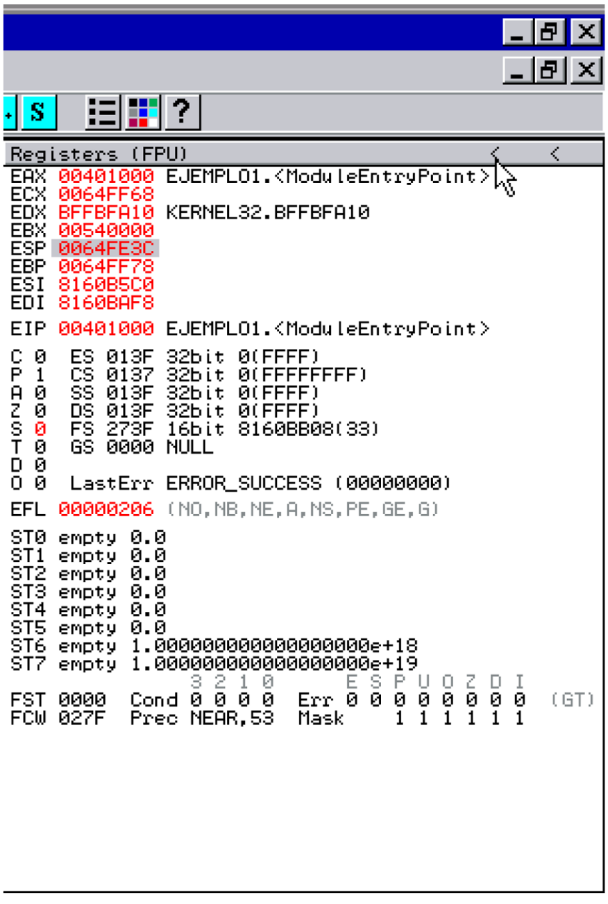
Aquesta és la finestra principal per a l'anàlisi d'aplicacions d'OllyDbg. També s'anomena *CPU Window*. Es pot cridar també amb la combinació de tecles "Alt + C". En aquesta finestra s'observen 5 seccions:

1. Desassemblatge. En aquesta zona apareix l'adreça de memòria de cada instrucció, el codi màquina corresponent, la seva traducció a llenguatge assemblador i una última part en la qual hi ha l'anàlisi que l'OllyDbg ha fet de l'aplicació carregada. Es pot veure l'anàlisi del codi fet per l'OllyDbg en la figura següent.



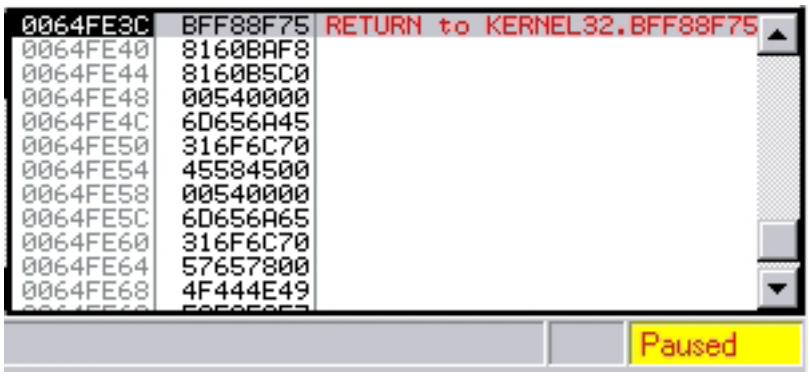
OllyBdg. Secció de desassemblatge de codi

2. Registres. En aquesta secció es pot observar l'estat dels registres en cada pas de l'execució del programa. Cada vegada que el programa avança en l'execució d'una o diverses línies de codi l'estat dels registres s'actualitza.



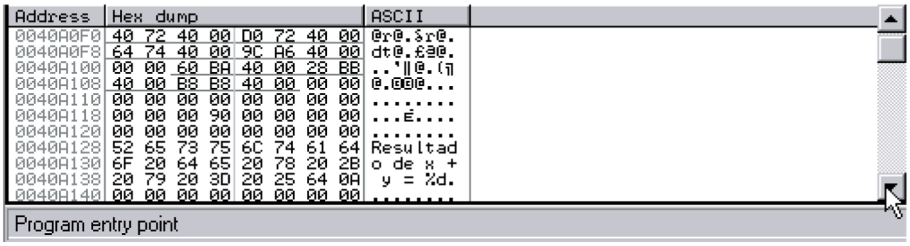
OllyDbg. Secció de registres

3. Pila. Igual que la resta de seccions, aquesta també s'actualitza a cada pas de l'execució. En aquesta secció es mostra el contingut de la pila. Habitualment, es mostra també una anàlisi del contingut de la pila en una columna situada a la dreta. S'hi poden veure fàcilment les adreces de retorn de les crides a funcions fetes amb *call*.



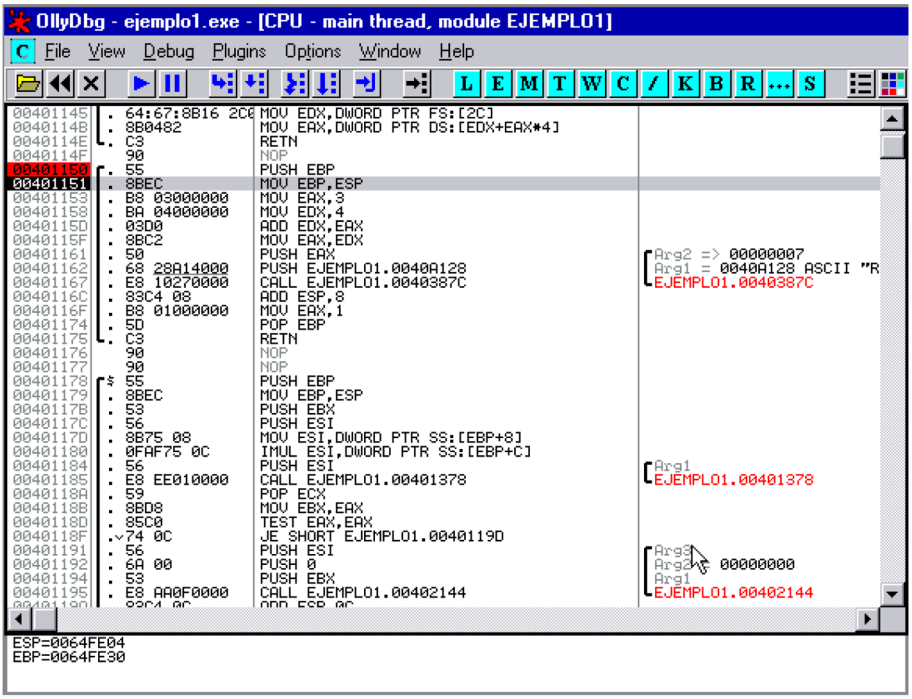
OllyDb. Secció de contingut de la pila

4. Buidatge de memòria. En aquesta secció es pot veure un buidatge del contingut de la memòria. És possible desplaçar-s'hi i fins i tot fixar el punt de la memòria que es vol observar.



OllyDbg. Secció de buidatge de memòria

5. Àrea d'ajuda. En aquesta secció s'ofereix informació sobre el que fan les ordres en assemblador. Les dades sobre els operands de l'ordre per executar són mostrades sempre que sigui possible. Això facilita la localització de la informació en un punt de l'execució, ja que no s'han de buscar cadascun dels operands en les diferents finestres d'informació. Les dades més rellevants en un moment concret es poden veure aquí, i si es requereix més informació, es pot observar el contingut de les altres finestres.



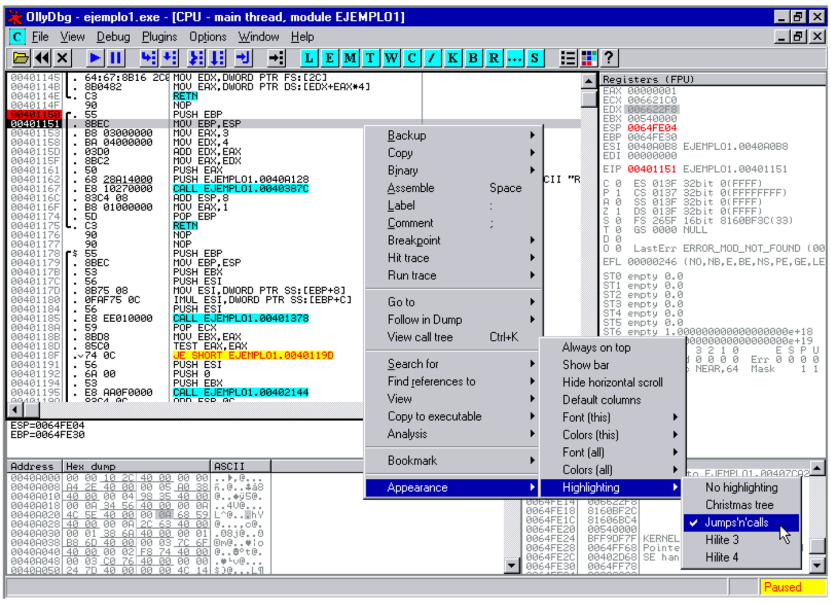
OllyDbg. Desassemblatge més informació

En la figura anterior es pot observar que es mostra informació sobre els registres que intervindran en l'execució de la instrucció següent en assemblador. La instrucció és `MOV EBP,ESP`, i en la part inferior es pot veure el contingut de tots dos registres just abans d'executar-se la instrucció.

1.2.2. Opcions

Aquesta aplicació és molt configurable. És possible canviar la mida de totes les seccions, a més del tipus de lletra i colors de codi.

> Per exemple, una de les configuracions de colors de codi que hi ha és la de marcar les instruccions relacionades amb crides a funcions i salts. D'aquesta manera, la lectura del codi es torna més visual.



OllyDbg. Opcions de la secció de codi (colors)

En la figura anterior es pot observar com es poden configurar les opcions per a mostrar dades o codi. Es pot apreciar que la finestra del codi assemblador té moltes opcions. De fet, s'obre un menú contextual fent clic amb el botó dret del ratolí en les diferents seccions de l'aplicació. Cada secció disposa de les seves pròpies opcions i les opcions mostrades depenen en tot moment de l'element seleccionat de la secció. Entre les opcions més interessants que es poden trobar en cada secció hi ha:

- Desassemblatge
  - Modificació del codi màquina.
  - Assemblatge d'instruccions a codi màquina.
  - Gestió de punts de ruptura.
  - Mostrar dades d'una adreça de memòria seleccionada, en la secció de buidatge de memòria.
  - Anar a un punt de l'execució del programa.
  - Veure l'arbre de crides.
  - Buscar una dada concreta.

- Registres
  - Possibilitat de modificar el contingut de tots els registres.
  - Visualització del grup de registres que volem.

- Pila
  - Possibilitat de modificar les dades de la pila.
  - Alterar la pila introduint o traient elements (`PUSH/POP`).
  - Mostrar dades d'una adreça de memòria seleccionada en la secció de buidatge de memòria o en la pila.

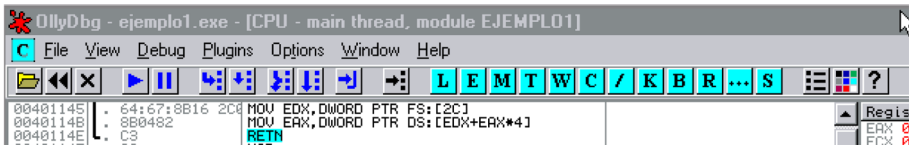
- Buidatge de memòria
  - Desassemblatge de posicions de memòria.
  - Buscar referències a dades.

– Gestió de punts de ruptura associats a l'accés a dades.
– Mostrar les dades en diferents formats (hexadecimal, ASCII, mida de les dades que es visualitzen...).

Aquestes no són totes les opcions, però són algunes de les més interessants.

El desassemblatge de posicions de memòria permet comprovar si un codi maliciós s'ha introduït en memòria de manera correcta perquè l'*exploit* tingui èxit.

Es disposa també d'operacions per a controlar l'execució del procés que s'està analitzant. De la mateixa manera que el GDB tenia una instrucció *run* per a arrencar el programa i `stepi` per a executar una instrucció en assemblador, en l'OllyDbg hi ha opcions semblants.



OllyDbg - Icones

Tenint en compte les icones que apareixen d'esquerra a dreta:

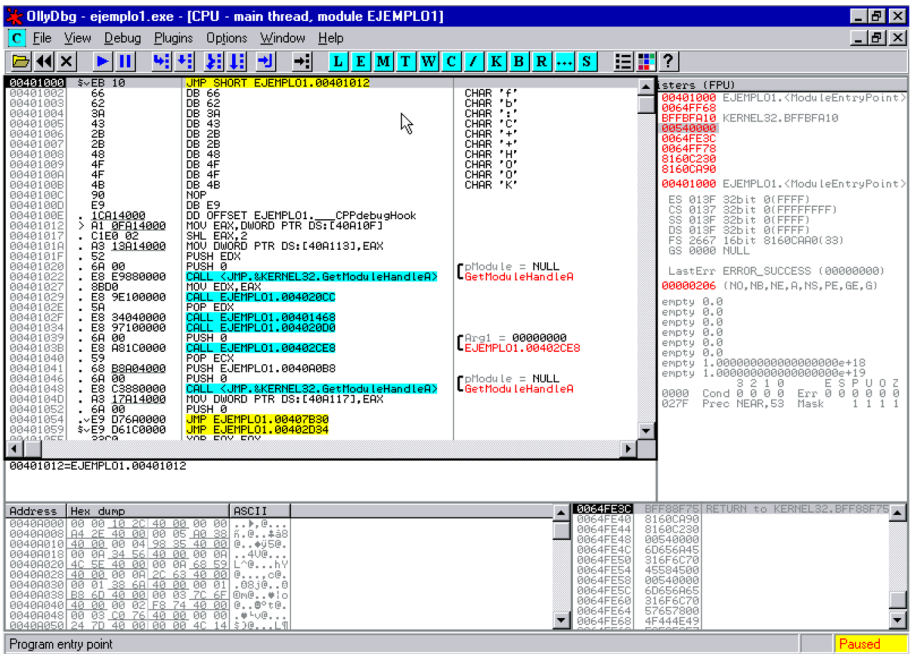
- Obrir arxiu (F3)
- Reprendre el programa (Ctrl+F2)
- Tancar el programa (Alt+F2)
- Executar (F9)
- Pausa en l'execució (F12)
- Executar una instrucció (F7)
- Executar una instrucció sense entrar en les crides (F8)
- Executar automàticament les instruccions (animació) entrant en les crides (Ctrl+F11)
- Executar automàticament les instruccions (animació) sense entrar en les crides (Ctrl+F12)
- Executar fins a sortida de funció (Ctrl+F9)
- Anar a una instrucció en el desassemblador
- Mostrar la finestra de "Log" (Alt+L)
- Mostrar la finestra de mòduls carregats (Alt+E)
- Mostrar la finestra de memòria (Alt+M)
- Mostrar la finestra de fils
- Mostrar la finestra de "Ventanas"
- Mostrar la finestra de CPU (Alt+C)
- Mostrar la finestra de "Patches" (Ctrl+P)
- Mostrar la finestra de crides (Alt+K)
- Mostrar la finestra de punts de ruptura (Alt+B)
- Mostrar la finestra de referències
- Mostra la finestra d'execució d'un *trace*
- Mostra la finestra del codi font

- Configuració d'opcions de depuració (Alt+O)
- Opcions d'aparença
- Ajuda

Es pot veure que la quantitat d'opcions és molt gran. En cada cas s'utilitzarà tan sols un subconjunt de totes les eines disponibles.

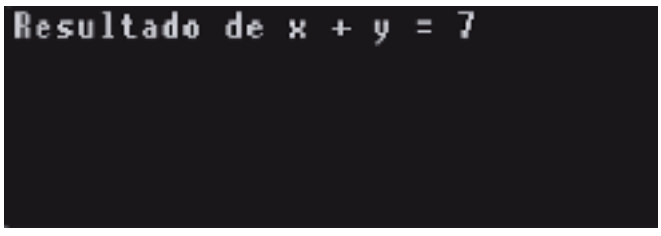
1.2.3. Modificació d'un programa en execució

Es procedeix a continuació a alterar el programa `exemple1` en temps d'execució, de la mateixa manera que s'ha fet anteriorment, perquè es mostri un valor diferent (123) per pantalla al final de l'execució. La primera acció que cal fer és carregar el programa en memòria per a poder procedir a analitzar-lo.



OllyDbg. *Exemple1* carregat

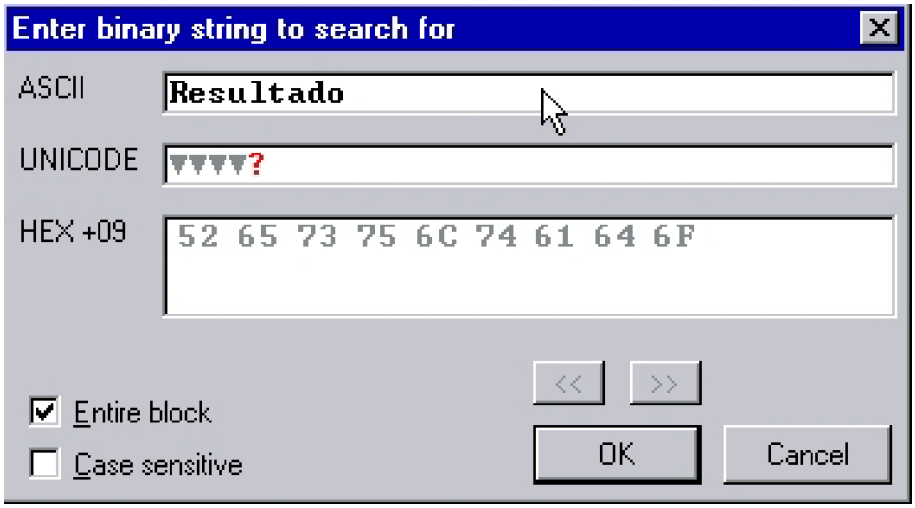
El pas següent consisteix a localitzar el punt en el qual es vol actuar. Del programa executat es coneix la sortida que dóna, després de l'execució:



OllyDbg. Resultat *exemple1*

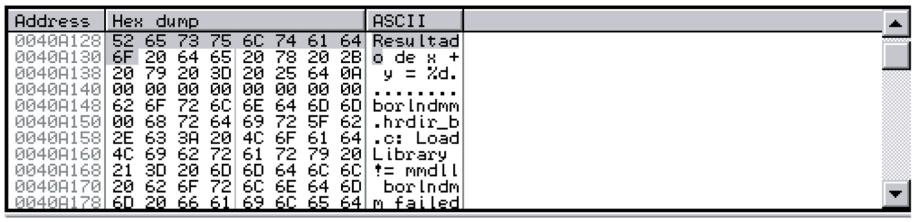
La cadena de caràcters `Resultat de...` és retornada pel programa com a sortida. Aquesta informació proporciona una idea del punt en el qual es pot actuar per a modificar el resultat. Una opció possible és buscar la cadena de caràcters en la memòria del programa. Es procedeix a fer clic amb el botó dret del ratolí sobre la secció de memòria, i seleccionem l'opció "Search for" i del

submenú "Binary string". (Es pot aconseguir el mateix resultat amb la combinació de tecles "Ctrl+B". Apareix una finestra en la qual es pot especificar el text per buscar.)



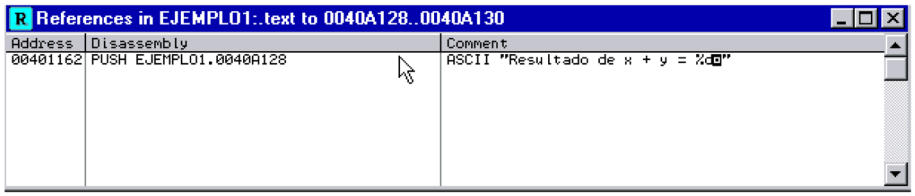
OllyDbg. Buscar text

L'execució d'aquesta sentència localitza en la memòria de l'aplicació el text. El resultat apareix en la secció de buidatge de memòria.



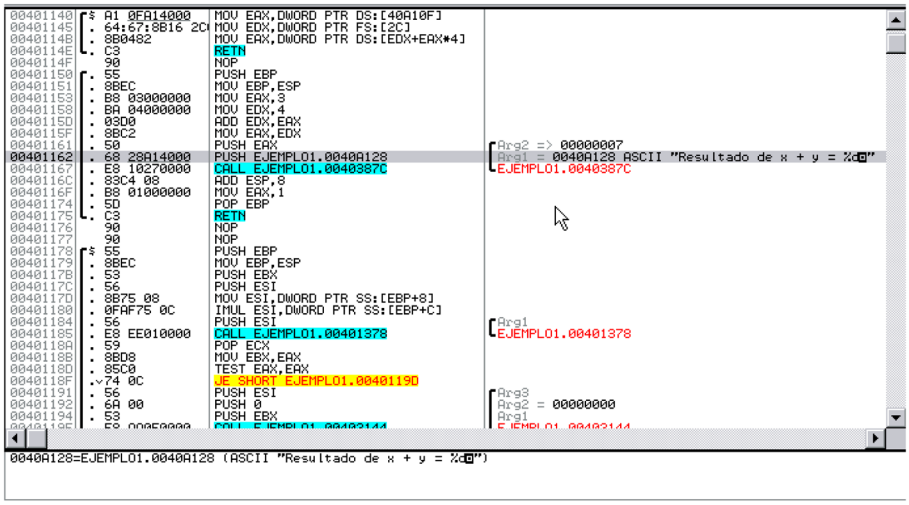
OllyDbg. Resultat de la cerca de text en memòria

Ara es necessita saber en quin punt de l'aplicació s'accedeix (o es fa referència) a aquesta dada. Molt probablement, en algun d'aquests punts hi deu haver la sentència que mostri per pantalla la informació que es vol alterar. Per a aconseguir-ho es fa clic amb el botó dret del ratolí sobre el primer caràcter de la cadena buscada i se selecciona l'opció "Find references". També es pot pressionar "Ctrl+R".



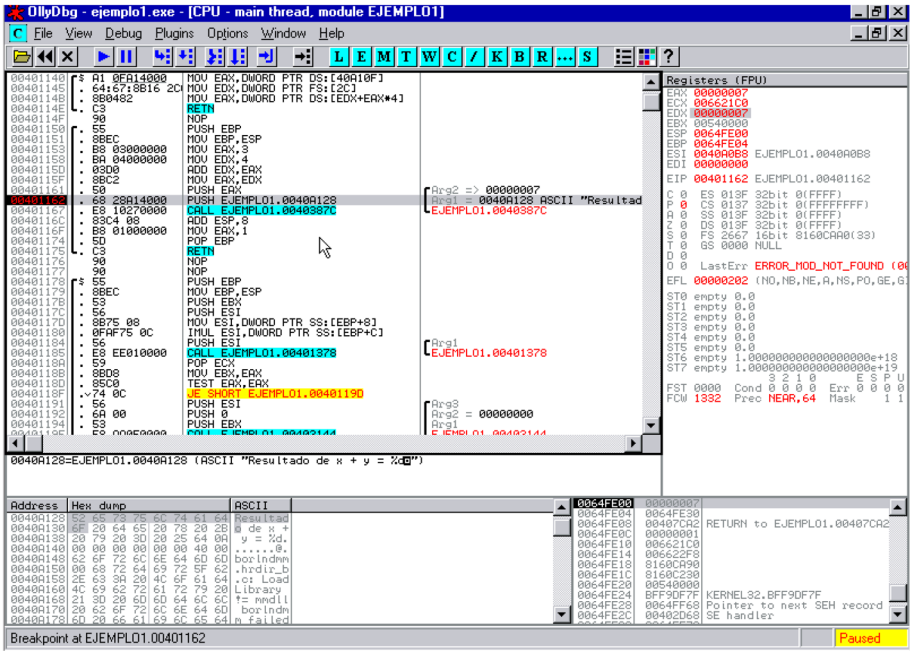
OllyGbd. Finestra de referències

El resultat mostra una línia de codi que introdueix en la pila una referència a la cadena. Fent clic amb el botó dret del ratolí es pot triar l'opció "Follow indisassembler", per a obrir la finestra de "CPU" i mostrar la línia de codi d'interès.



OllyDbg. Codi que fa referència a la cadena buscada.

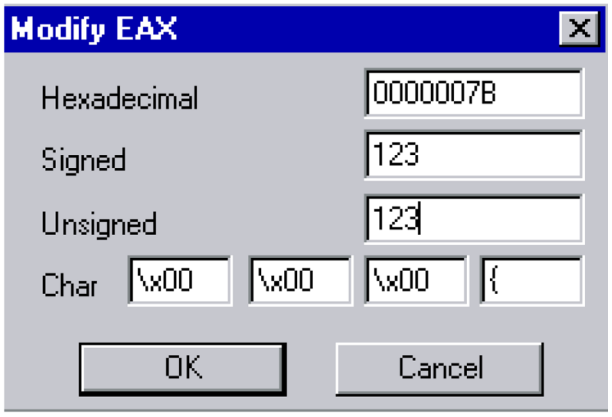
Es pot observar que hi ha una crida a una funció (de la qual no es coneixen més dades que l'adreça), a la qual es passen dos paràmetres en la pila. Un d'enter i una cadena de caràcters, que té un format molt semblant al que usa la instrucció `printf` per a mostrar text per pantalla. La cadena de caràcters conté el text `%d`, que en `printf` s'interpreta com el comodí per a mostrar un nombre enter. Aquest nombre podria ser el segon paràmetre. Tenint això en compte, es procedeix a marcar la línia amb un punt de ruptura i a executar el programa. Un punt de ruptura es pot crear fent doble clic en el codi màquina de la línia que volem. L'execució s'aconsegueix pressionant "F9".



OllyDbg. Punt de ruptura abans de la crida a una suposada funció `printf`

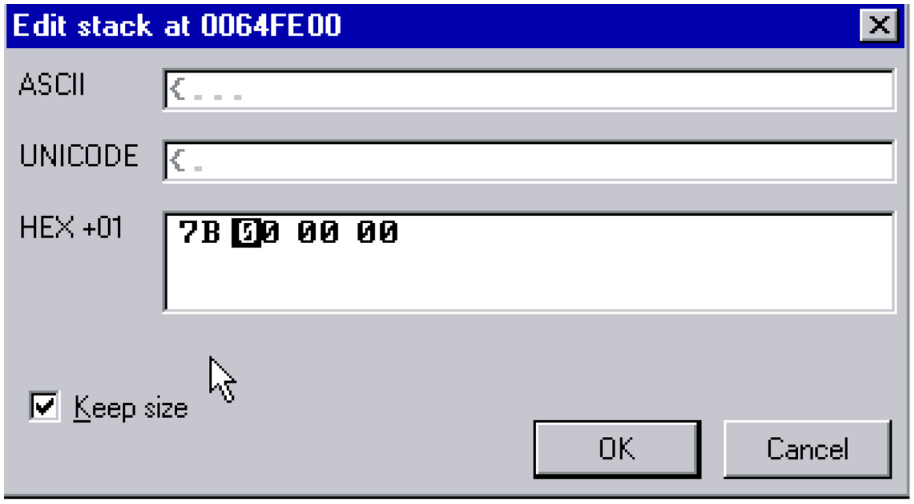
En la imatge s'observa com s'ha detingut l'execució en el punt seleccionat. El valor d'EAX és 7, cosa que concorda amb la sortida del programa. Per a alterar-ne l'execució cal canviar-ne el valor al nou valor que volem. Com que ja s'ha introduït aquest valor en la pila, s'haurà de canviar també allà. Si s'hagués

posat el punt de ruptura abans d'introduir el valor d'EAX en la pila, el treball de canviar el valor de la pila no hauria estat necessari. Es procedeix a canviar el valor del registre EAX fent doble clic a sobre:



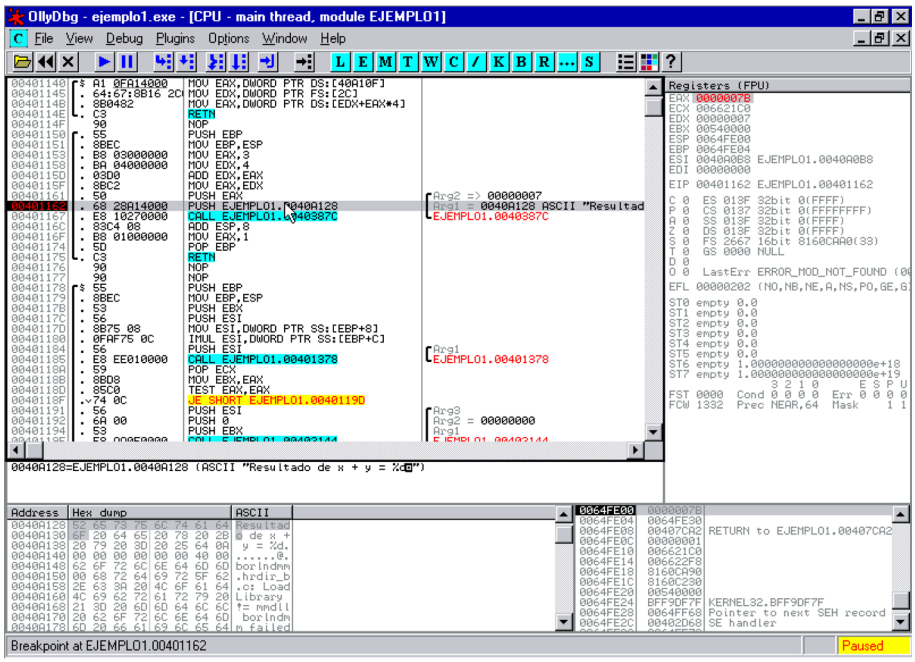
OllyDbg. Canvi d'EAX

En la figura se li assigna el nou valor `123`. Es procedeix ara a canviar el valor introduït en la pila, com a resultat d'haver executat un *push* del registre EAX. Es fa clic amb el botó dret del ratolí sobre la dada de la pila que es vol canviar i se selecciona l'opció "Edit" (Ctrl+E).



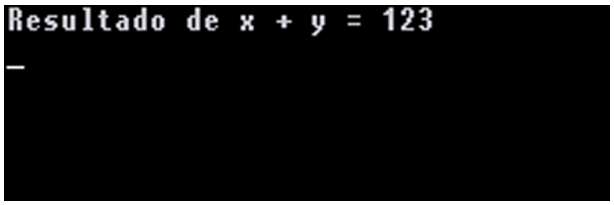
OllyDbg. Editar una dada de la pila

En aquest cas és necessari fer el canvi en hexadecimal. Es canvia el valor i es desa. El resultat mostraria la informació següent en pantalla:



OllyDbg. Resultat modificat

Una vegada ja estan acabats tots els detalls del canvi que es vol fer, es pot procedir a l'execució del codi i veure si així s'ha obtingut el resultat que volem. Es pressiona dues vegades "F8" (*step over*) per a executar la funció que "probablement" mostri la informació d'interès per pantalla.



OllyDbg. Resultat final

Com es pot observar, s'ha aconseguit el resultat que volem, perquè ara el programa no ha mostrat el 7 esperat sinó el `123` que volem.

1.2.4. Conclusions

Aquest depurador és molt visual i ofereix un entorn bastant bo per a introduir-se en el món del codi màquina i assemblador. Es disposa de molta informació accessible tan sols mirant a una altra part de la finestra activa. Evidentment, no hem vist totes les opcions disponibles en aquesta aplicació, ja que l'interès d'aquest mòdul és introduir l'eina per a poder utilitzar-la en altres casos que proposarem en mòduls posteriors. L'eina fa la inspecció d'un sistema a baix nivell durant l'execució d'un programa, i ofereix un entorn perfectament usable per a la realització d'*exploits*.

2. Compiladors/llenguatges

Hi ha multitud de compiladors i llenguatges de programació en el mercat. Tots tenen avantatges i inconvenients. Entre tots, els més utilitzats per a la generació d'*exploits* i cerca de vulnerabilitats són tres.

- Assemblador
- C
- Perl

Com hem vist, el llenguatge assemblador s'usa per a l'anàlisi d'aplicacions a baix nivell durant l'execució d'aquestes. És el llenguatge que més a prop es troba de les instruccions que executa la màquina (codi màquina). Això ofereix moltes possibilitats i opcions a l'hora de buscar o explotar vulnerabilitats en programes.

El llenguatge de programació C s'usa habitualment per la seva flexibilitat a l'hora de programar determinades aplicacions i pel seu rendiment. Els programes escrits en aquest llenguatge destaquen pel seu elevat rendiment. És molt freqüent l'ús d'aquest llenguatge per a programar *exploits*.

En alguns casos també s'usa Perl per a la realització d'*exploits*, encara que el més habitual és que aquests estiguin escrits en C. Normalment l'ús d'*exploits* està associat a entorns de tipus Unix, encara que no necessàriament ha d'estar limitat a aquests entorns.

Tenint en compte tot el que hem vist fins ara, resulta interessant veure com es pot obtenir codi màquina a partir d'una sèrie de sentències en assemblador. D'aquesta manera es podrien modificar apropiadament les instruccions que ha d'executar el processador en un programa (canviar el codi durant l'execució mitjançant un compilador) o crear *shellcodes* a mida. La creació de *shellcodes* la veurem en un altre mòdul del curs, però és interessant conèixer com es pot obtenir el codi màquina a partir d'assemblador.

Modificar una aplicació durant l'execució i fer aquests canvis permanents en el fitxer executable perquè l'aplicació faci una sèrie de canvis introduïts per l'usuari és el que es coneix com a *cracking*. Les eines vistes fins ara es poden utilitzar amb aquest propòsit, encara que no és l'objectiu d'aquest curs.

2.1. Generació d'*opcodes*

Per a obtenir el codi màquina a partir d'una sèrie de sentències escrites en codi màquina es pot crear un programa en assemblador i compilar-lo. Hi ha diversos assembladors en el mercat, com el NASM[4] o AS[4].

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

Es considera el programa següent en assemblador:

```
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 fa cap operació concreta, però il·lustra el cas de necessitar el codi màquina corresponent a una sèrie d'operacions en aquest codi. Es pot assemblar el programa anterior amb l'AS:

```
as assemblador.s
```

El resultat, desassemblat el codi màquina anterior amb *objdump*, és:

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

Utilitzant el GDB es poden obtenir els codis hexadecimals (*opcodes*) de manera gairebé immediata per a usar-los 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...
```