

Introducción a depuración en Linux

Iván Mosquera Paulo ktulur@gmail.com

Iker Castaños hackercasta@gmail.com

18/12/2007

Índice de contenido

Depuradores modo usuario	2
Introducción: entendiendo la llamada al sistema ptrace.....	2
EJEMPLO #1.....	3
EJEMPLO #2.....	4
STRACE.....	7
EJEMPLO #1.....	7
EJEMPLO #2.....	8
LTRACE.....	10
GDB.....	12
Un frontend de GDB : KDBG.....	13
UML (user-mode-linux).....	17
Depuradores de modo kernel.....	19
Fuentes y enlaces relacionados.....	20

Depurar/Debugging: identificar y corregir errores de programación.

Nosotros vamos a **ver un poco los distintos depuradores disponibles para ver cómo funciona la ejecución de nuestros programas de ASO y ver en la práctica la teoría dada**. No vamos a comentar por tanto otros aspectos que habría que tener en cuenta para que el uso de depuradores sea verdaderamente eficaz en la corrección de errores de software.

Uso general:

Normalmente nosotros ejecutamos el depurador y el depurador a su vez crea un proceso hijo que con una llamada al sistema “exec” pasa a ser el programa que nosotros queremos depurar.

Depuradores modo usuario

Introducción: entendiendo la llamada al sistema ptrace

Se trata de depuradores que son **programas a nivel de usuario**. Hacen uso de la llamada al sistema **ptrace** . A continuación vemos un poco como funciona esta llamada al sistema. (Esta parte es la menos práctica y si se prefiere puede saltarse a la siguiente : STRACE)

El algoritmo para hacer uso de ptrace suele ser algo así:

1. Un proceso padre crea un hijo y el hijo hace la llamada ptrace para permitir al padre controlarle.
2. El padre usa la llamada al sistema “wait” esperando así a un evento que cambie el estado del proceso hijo.
3. Al ocurrir el evento el kernel despierta al padre. Lo que retorne “wait” indica que el hijo se ha parado y da información sobre qué ha causado la parada.
4. Entonces el padre controla el hijo con las operaciones pasadas a “ptrace”.

La sintaxis de **ptrace()** es la siguiente:

int ptrace (int cmd, int pid, int addr, int data)

cmd: diferentes operaciones posibles.

pid: pid del proceso.

addr: posición en el espacio de direcciones del proceso.

data: su sentido depende de cmd.

EJEMPLO #1

Nota: Para compilarlos basta con hacer “gcc nombreprograma.c” pero es conveniente utilizar “-o nombreprograma” para especificar el nombre del ejecutable en lugar de “a.out”.

Tenemos el siguiente programa donde simplemente hacemos un loop:

```
#include <sys/ptrace.h>
main()
{
    while(1);
}
```

Este programa lo ejecutamos y lo podemos matar en la misma terminal con Control-C por ejemplo. Ahora veamos qué hace si ponemos la sentencia ptrace().

```
#include <sys/ptrace.h>
main()
{
    ptrace(PTRACE_TRACEME, 0, 0, 0);
    while(1);
}
```

Ahora en la ejecución veremos que si tratamos de terminarlo con Control-C lo que conseguiremos en su lugar es pararlo:

```
ktulur@amilopro:~/ktulur-svn/euitiBI-itig/3/ASO/programas/docudebuggers$ ./a.out
[1]+  Stopped                  ./a.out
```

Si vemos el man de ps veremos que para el estado “T” que es el estado en el que nos aparecerá “./a.out” al hacer ps es “Stopped, either by a job control signal or because it is being traced.”

El programa se va parando en cada instrucción. Como en este caso no hay un proceso padre que pueda hacer algo al respecto nuestro programa simplemente pasa a estar parado en la primera instrucción.

EJEMPLO #2

Ahora veremos un ejemplo más complicado en el que lo que queremos es **contar el número de instrucciones que ejecuta el hijo**. Esto lo podemos hacer gracias a que con `ptrace` vamos parando al hijo en cada instrucción y a su vez el hijo va despertándose cada vez para incrementar el contador:

Nota: Puede interesarnos **compilarlo como estático** (en lugar de dinámico) ya sea para mayor rendimiento o para **poder usar el ejecutable en otro equipo sin preocuparnos por las librerías no instaladas**. Para entendernos:

Hasta ahora lo habríamos compilado así: `gcc programa.c -o programa`

Si hacemos un “`ldd programa`” veremos lo siguiente:

```
linux-gate.so.1 => (0xffffe000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7ddd000)
/lib/ld-linux.so.2 (0xb7f38000)
```

Son las librerías de las que depende dinámicamente.

Ahora proponemos compilarlo así: `gcc -static programa.c -o programa`

Si hacemos `ldd` veremos lo siguiente:

`ldd`: saliendo con un código de salida desconocido (126)

Como pega obvia de compilarlo estáticamente es que el **tamaño es muchísimo mayor** al tener que incluir las librerías necesarias en el propio ejecutable en lugar de simplemente linkarlas.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <syscall.h>
#include <sys/ptrace.h>

void main (void) {

    long long contador = 1; // contador de instrucciones
    int estado;           // valor de retorno del hijo
    int pid;              // pid del hijo.

    /* 1a. Se crea proceso hijo */
    if ((pid=fork())==-1) {
        perror("error");
        _exit(-1);
    } else {
        if (pid==0) {
```

```

//Aquí entra el hijo
/* 1b. El hijo hace la llamada ptrace para permitir al padre controlarle. Existe un
mecanismo alternativo mediante */
ptrace(PTRACE_TRACEME,0,0,0);
/* Se hace exec del programa a depurar */
execl("/bin/ls","ls",NULL);

/* Termina el hijo */
} else {
//Aquí entra el padre.
/* 2. El padre usa la llamada wait esperando así a un evento que cambie el estado del
proceso hijo */
wait(&estado);
if (ptrace(PTRACE_SINGLESTEP,pid,0,0) != 0) printf("ptrace");
wait(&estado);

while (estado == 1407) {
    contador++;
    if (ptrace(PTRACE_SINGLESTEP,pid,0,0) != 0) perror("ptrace");
    wait(&estado);
}
}
}

printf("Contador : %d\n", contador);
}

```

Veremos como al ejecutar el programa en el directorio actual siempre devuelve el mismo contador mientras que si probamos a ejecutar el programa en otro lugar el contador es distinto. Esto es así ya que el hijo hace un “exec” de “ls” y por tanto, el número de instrucciones del hijo va a depender del número de ficheros del directorio actual.

El ejemplo visto sólo representa la punta del iceberg de lo que podemos hacer con ptrace(). Por ejemplo, **con ptrace() es posible ver las llamadas al sistema del proceso hijo e incluso manipular los registros y la ejecución del proceso hijo.**

Sin embargo **en la práctica no suele utilizarse ptrace() en los programas para depurarlos sino que ya hay gente que se ha preocupado de implementar programas que reciben el programa a depurar y realizan los ptraces correspondientes y según una serie de opciones que se les pase.**

De modo que **todos los depuradores de Linux de modo usuario están basados en la llamada al sistema ptrace.**

Nota: Con ptrace solemos poder sacar los códigos de las llamadas al sistema, pero no el identificador como estamos acostumbrados a ver.

Podemos ver los códigos de las llamadas al sistema en `/usr/include/asm-i386/unistd.h`

STRACE

No es un depurador propiamente dicho ya que le faltan muchas funcionalidades para serlo pero tiene sus propios usos típicos para los cuales es más cómodo que un depurador.

STRACE es un programa que nos **permite ver las llamadas al sistema y las señales de otro programa que le pasemos por parámetro.**

Veamos un par de ejemplos para entender su uso.

EJEMPLO #1

Compilemos estáticamente el siguiente programa (`gcc -static programa.c -o programa`):

```
#include <stdio.h>
main(){
    printf("Hola mundo\n");
}
```

Ahora ejecutamos:

`strace ./programa`

```
execve("./programa", ["/programa"], [/* 36 vars */]) = 0
uname({sys="Linux", node="amilopro", ...}) = 0
brk(0) = 0x80be000
brk(0x80becb0) = 0x80becb0
set_thread_area({entry_number:-1 -> 6, base_addr:0x80be830, limit:1048575, seg_32bit:1,
contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
brk(0x80dfcb0) = 0x80dfcb0
brk(0x80e0000) = 0x80e0000
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 5), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0xb7fa0000
write(1, "Hola mundo\n", 11Hola mundo
) = 11
exit_group(11) = ?
Process 9818 detached
```

Lo que **nos muestra la salida** de `strace` son las **llamadas al sistema en un formato de lenguaje C**. Esto es así **independientemente del lenguaje en el que esté el código** de nuestro programa, no es más que el formato que utiliza `strace` para darte la información.

Así puedes encontrar unas cuantas llamadas al sistema como “`execve`” cuando se crea el proceso, o “`write`” cuando escribe en pantalla.

Hemos sugerido compilarlo estáticamente ya que de lo contrario la salida es mucho más grande ocupando mucho más este documento al mostrarse llamadas al sistema que acceden a las librerías.

Sin embargo, es interesante que pruebes por tu cuenta a hacer strace de un programa dinámico. Verás al principio llamadas al sistema “open” y “access” a ficheros “.so”. Esos son los accesos a las librerías con las que está enlazado.

EJEMPLO #2

Ahora otro ejemplo pero con dos procesos:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main()
{
int ret;
int estado;
ret=fork();
if (ret==-1) { printf("error del fork"); _exit(-1);}
else if (ret==0) { printf("soy el hijo. Mi pid es %8d \n",getpid()); }
else { wait(&estado);
printf("soy el padre.Mi pid es %8d \n",getpid());
}
return 0;
}
```

Lo compilamos estáticamente y vemos la salida de “[strace programa](#)”

```
execve("./a.out", ["/a.out"], [/* 35 vars */]) = 0
uname({sys="Linux", node="amilopro", ...}) = 0
brk(0) = 0x80be000
brk(0x80becb0) = 0x80becb0
set_thread_area({entry_number:-1 -> 6, base_addr:0x80be830, limit:1048575, seg_32bit:1,
contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
brk(0x80dfcb0) = 0x80dfcb0
brk(0x80e0000) = 0x80e0000
clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0x80be878) = 9996
soy el hijo. Mi pid es 9996
--- SIGCHLD (Child exited) @ 0 (0) ---
```

```
wait4(-1, [{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0, NULL) = 9996
getpid() = 9995
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0xb7f6e000
write(1, "soy el padre.Mi pid es 9995 "..., 33soy el padre.Mi pid es 9995
) = 33
exit_group(0) = ?
Process 9995 detached
```

Podemos ver algunas llamadas al sistema que hemos visto al estudiar procesos como:

clone

getpid

write

El resto de las **llamadas al sistema puedes saber qué hacen con ayuda del comando “man nombre_llamada”**.

Verás que no se ven las llamadas al sistema del proceso hijo. Esto es así porque **para ver las llamadas al sistema de procesos hijos debes utilizar el parámetro “-f” de strace**.

Puedes seguir mirando las llamadas del sistema que realiza cualquier otro programa desde “ifconfig” hasta “oowriter”. Verás que cuanto más complejo es el programa más llamadas al sistema y señales suele tener, y por tanto la salida de strace se hace enorme.

STRACE dispone de muchas más opciones. Para verlas “man strace”. Por ejemplo **para ver las llamadas del sistema relacionadas con cosas de redes** al hacer un ping podemos hacer:

```
strace -e trace=network ping google.es
```

O **para ver únicamente las llamadas al sistema “open()”** de un ifconfig:

```
strace -e trace=open ifconfig
```

Veremos que accede tanto a ficheros de librerías como a cosas en /proc.

Por último comentar que **strace puede adherirse a un programa ya en ejecución con el parámetro “-p pid”**.

LTRACE

LTRACE muy similar en el uso a STRACE. La diferencia reside en que en lugar de mostrarnos las llamadas al sistema y las señales, nos muestra la llamadas a librerías.

Veamos un ejemplo:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main()
{
int ret;
int estado;
ret=fork();
if (ret==-1)
    {printf("error del fork");}
else if (ret==0)
    {printf("soy el hijo. Mi pid es %8d \n",getpid());}
else
    {wait(&estado);
printf("soy el padre.Mi pid es %8d \n",getpid());
printf("el estado del hijo es %d \n",estado);}
return 0;
}
```

Lo compilamos con

```
gcc programa.c -o programa
```

Si lo compilamos estáticamente veríamos que LTRACE nos da error, lo cual tiene sentido si pensamos en lo que hace LTRACE.

Hacemos LTRACE del ejecutable:

```
ltrace ./programa
```

y lo que nos mostrará es lo siguiente:

```
ktulur@amilopro:~/ktulur-svn/euitiBI-itig/3/ASO/programas/sistemaprocesos$ ltrace ./1
__libc_start_main(0x8048414, 1, 0xbfc163b4, 0x80484b0, 0x80484a0 <unfinished ...>
fork(soy el hijo. Mi pid es 6692
<unfinished ...>
--- SIGCHLD (Child exited) ---
```

```
<... fork resumed> )           = 6692
wait(0xbfc16320)                = 6692
getpid()                        = 6691
printf("soy el padre.Mi pid es %8d \n", 6691soy el padre.Mi pid es 6691
) = 33
printf("el estado del hijo es %d \n", 0el estado del hijo es 0
) = 25
+++ exited (status 0) +++
```

Vemos que lo que nos muestra son las llamadas a funciones. No hay que olvidar que aunque muchas funciones que usamos en C tienen nombres coincidentes con algunas llamadas del sistema, no estamos invocando las llamadas al sistema directamente desde nuestro programa sino que lo hacemos a través de las librerías. Esto no ocurre con programas en lenguaje ensamblador donde sí que se realizan las llamadas directamente.

Aunque la salida que nos da LTRACE es más cómoda de leer, lo cierto es que da mucha menos información que STRACE.

GDB es el depurador oficial de proyecto GNU.

Nos permite hilar mucho más fino que con STRACE. Veremos algunos casos de uso que pueden ser interesantes.

Ver un core dump.

El core dump es un vuelco a memoria secundario en un fichero “core” de la imagen del proceso tal y como estaba al ser abortado.

En Linux esta funcionalidad está desactivada por defecto y se activa con :

```
ulimit -c unlimited
```

Hagamos un programa que vaya a fallar seguro para generar luego un core dump y verlo con GDB:

```
#include <stdio.h>
main() {
printf("%d",5/0);
}
```

Lo compilaríamos normalmente con `gcc 1.c -o 1` pero en este caso vamos a utilizar además un **parámetro adicional “-ggdb”** que lo que hace es meter en el ejecutable información adicional que puede ser útil para GDB como puede ser el propio código fuente. Podría decirse que lo compilamos en modo DEBUG.

y lo ejecutamos:

```
root@amilopro:/tmp# ./1
```

Exepción de coma flotante (core dumped)

¿Qué hacemos ahora con ese fichero “core” que nos ha generado en el directorio actual?

Pues con la sintaxis “`gdb -q nombre_programa core`”, gdb nos dará la causa.

En nuestro ejemplo

```
Core was generated by `./1'.
```

```
Program terminated with signal 8, Arithmetic exception.
```

```
#0 main () at 1.c:7
```

```
7 printf("%d",5/0);
```

Como puedes ver, GDB nos da información bastante útil como la función en dónde estaba ejecutándose así como la sentencia que ha fallado.

GDB es un programa interactivo (como GRUB) y para salir usamos “quit”. Hay muchos otros comandos como por ejemplo para definir puntos de ruptura, avanzar o retroceder instrucciones, imprimir valores de variables, hacer backtrace... Podemos ver cada uno con “`man gdb`”. En este documento utilizaremos todos estos comandos desde un aplicación frontend de GDB.

Un frontend de GDB : KDBG

Se trata de uno de los muchos “frontends” existentes para GDB (hay IDEs que tienen soporte para GDB también). Este en concreto pertenece al proyecto KDE. Para instalarlo al menos desde la distribución Ubuntu basta con:

[apt-get install kdbg](#)

Para ver en él algo con sustancia compilaremos un programa distinto:

```
#include <stdio.h>

main()
{
    int *p;
    int i;
    int mivalor;

    mivalor=0;
    printf("Esperamos tres segundos...\n");
    sleep(3);
    printf("Fin del sleep.");

    for(i=0;i<=6;i++) {
        mivalor = mivalor + i;
    }

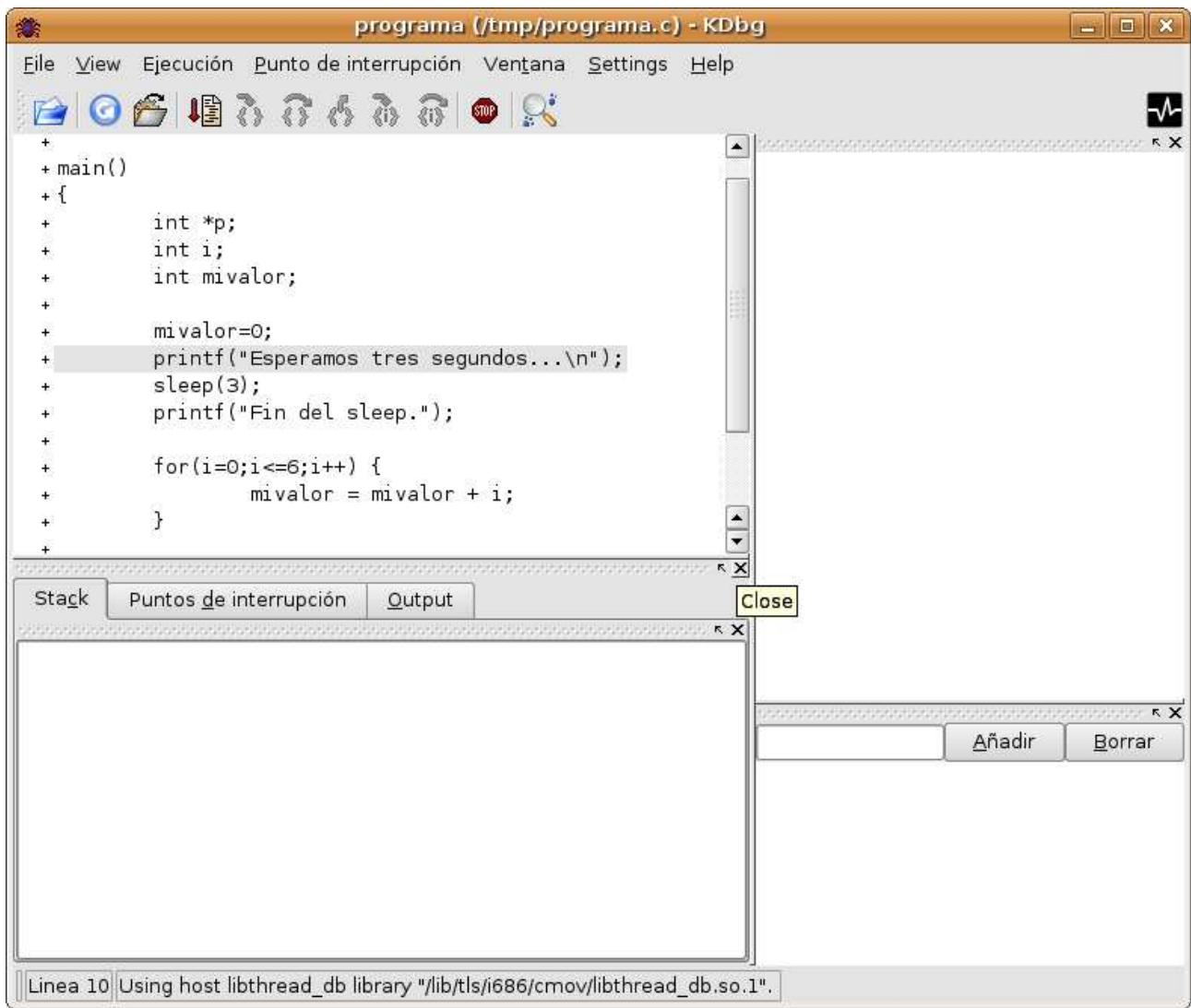
    p=23;
    *p=380000000;
    _exit(0);
}
```

[root@amilopro:/tmp# gcc -ggdb programa.c -o programa](#)

Ahora ejecutamos kdbg.

[root@amilopro:/tmp# kdbg programa](#)

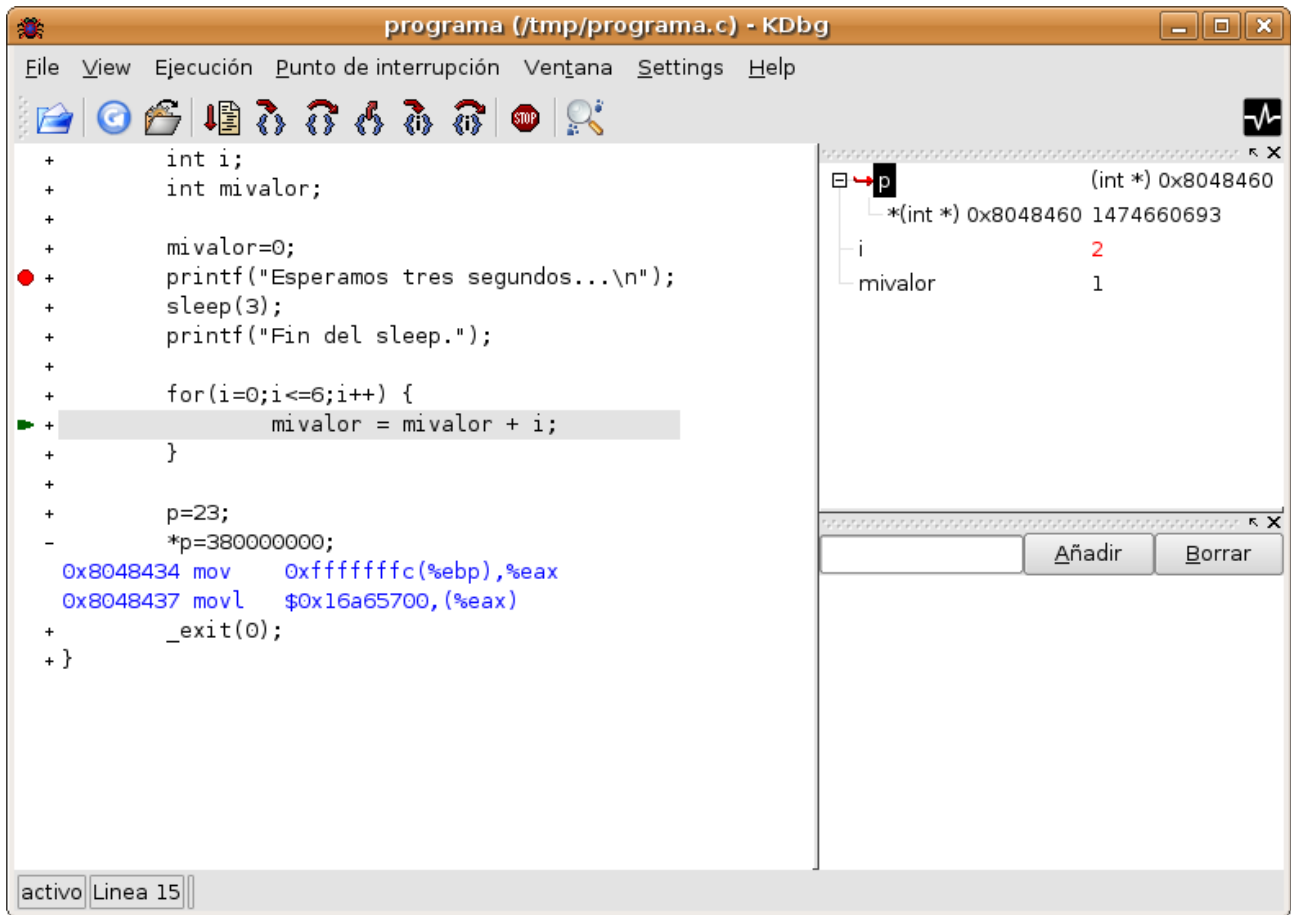
Nos saldrá una ventana como la siguiente. No nos debe de sorprender que aparezca el código fuente, no se trata de que lo haya conseguido con GDB ni mucho menos (a lo sumo puede conseguir el desensamblado) sino que debido al parámetro “-ggdb” hemos metido el código fuente en la tabla de símbolos del ejecutable.



Haciendo click en los “+” (algunos al menos) veremos que nos expande el código en ensamblador equivalente a la sentencia en C correspondiente.

Ahora lo que debemos hacer es definir un punto de ruptura. Para ello hacemos click en el botón derecho del ratón, se nos desplegará un menú contextual y hacemos click en la opción “Poner/Quitar puntos de interrupción”. Nos marcará la sentencia con un círculo rojo.

En la captura que hay a continuación vemos que hemos puesto un **punto de ruptura** (break point) y que hemos ido **avanzando la ejecución paso a paso**. La sentencia actual es la que está marcada con una flecha verde. Hemos ido avanzando haciendo click al botón “Otra instrucción” de la barra de herramientas. Ahora mismo estamos en el bucle y por tanto aunque sigamos dando a “Otra instrucción” seguimos dando vueltas al bucle hasta que se cumpla la condición de salida.



Hay una serie de ventanas que podemos activar/desactivar (Menú->View). En el pantallazo anterior vemos por ejemplo la ventana de variables locales. Si vamos avanzando instrucciones veremos como *i* va aumentando hasta 7 y así se sale del bucle. Por otra parte la variable *mivalor* irá acumulando *i* hasta valer 21,

“*p*” es un puntero a enteros que hacia el final se le asigna como contenido de la dirección de memoria a la que apunta el valor 23.

programa (/tmp/programa.c) - KDbg

File View Ejecución Punto de interrupción Ventana Settings Help

```
+ main()
+ {
+     int *p;
+     int i;
+     int mivalor;
+
+     mivalor=0;
+ 0x80483e4 movl  $0x0,0xffffffff4(%ebp)
+ ● +     printf("Esperamos tres segundos...\n");
+     sleep(3);
+ 0x80483f7 movl  $0x3,(%esp)
+ 0x80483fe call  0x8048320 <sleep@plt>
+     printf("Fin del sleep.");
+ 0x8048403 movl  $0x8048530,(%esp)
+ 0x804840a call  0x8048310 <printf@plt>
+
+     for(i=0;i<=6;i++) {
+         mivalor = mivalor + i;
+     }
+
+     p=23;
+ 0x804842d movl  $0x17,0xffffffffc(%ebp)
+ ● +     *p=380000000;
+     _exit(0);
+ }
```

Stack window:

- p (int *) 0x17
- *p (int *) 0x17 Cannot access memory at 0x17
- i 7
- mivalor 21

Buttons: Añadir, Borrar

activo | Linea 19

UML (user-mode-linux)

Existe un proyecto llamado **UML (user-mode-linux)** que **permite ejecutar un kernel Linux (modificado especialmente) como un proceso más del kernel de Linux que estamos corriendo actualmente**. Esto tiene como ventaja que ese kernel se ejecuta en nuestro sistema **como un proceso más a nivel de usuario**, al mismo nivel que cualquier otro y por tanto, podemos utilizar GDB por ejemplo para ver el funcionamiento del kernel , diagnosticar problemas con drivers etc..

Además tiene otras aplicaciones como la creación de honeypots, ejecución de servicios de red de manera aislada ...

Instalación:

Lo cierto es que es bien sencillo. De la siguiente URL: <http://user-mode-linux.sourceforge.net/> nos bajaremos dos ficheros: un kernel Linux UML y una imagen de sistema de ficheros (de la distribución Fedora concretamente). En la misma página WEB se dan instrucciones precisas pero vamos a mostrar a continuación como lo hemos hecho nosotros.

Hemos bajado los ficheros con WGET.

wget <http://user-mode-linux.sourceforge.net/linux-2.6.24-rc2.bz2>

wget http://uml.nagafix.co.uk/FedoraCore5/FedoraCore5-x86-root_fs.bz2

Los descomprimimos

```
bzip2 *.*
```

Ahora es cuando ejecutamos el Linux. Es necesario pasarle ciertos parámetros.

```
./linux-2.6.24-rc2 ubda=FedoraCore5-x86-root_fs mem=128M
```

Aquí las primeras líneas de su ejecución:

```
Core dump limits :
  soft - 0
  hard - NONE
Checking that ptrace can change system call numbers...OK
Checking syscall emulation patch for ptrace...OK
Checking advanced syscall emulation patch for ptrace...OK
Checking for tmpfs mount on /dev/shm...OK
Checking PROT_EXEC mmap in /dev/shm/...OK
Checking for the skas3 patch in the host:
- /proc/mm...not found: No such file or directory
- PTRACE_FAULTINFO...not found
- PTRACE_LDT...not found
UML running in SKAS0 mode
Linux version 2.6.24-rc2 (jdike@tp.user-mode-linux.org) (gcc version 4.1.2 20070925 (Red Hat 4.1.2-27)) #17 Mon
Nov 12 12:41:28 EST 2007
```

Finalmente nos aparecerá el prompt:

```
Fedora Core release 5 (Bordeaux)
```

Kernel 2.6.24-rc2 on an i686

localhost login:

En las ejecuciones sucesivas si vemos que nos da “kernel panic” al tratar de montar el sistema de ficheros es porque hemos dejado procesos creados por este Linux UML por ahí. Los matamos y fin del problema.

Ahora podríamos probar a ejecutarlo con STRACE y veríamos qué pasos va dando el kernel.

Depuradores de modo kernel

Sin el uso de UML para depurar o ver la ejecución del Kernel o de cualquier módulo o driver del kernel, es necesario el uso de Depuradores de modo kernel. Estos depuradores son **programas que se asisten de módulos (LKM) del kernel y por tanto están al mismo nivel que los drivers o cualquier proceso del kernel**, pudiendo acceder prácticamente a todo, y hacer cualquier cosa.

Este tipo de depuradores son muy utilizados en Windows para realizar ingeniería inversa de programas, diagnóstico de problemas con el hardware, “cracking”... Softice, Windbg o Syser son algunos ejemplos.

En Linux tenemos algunos depuradores de este tipo que pueden activarse al compilar el kernel. Además existen algunos programas análogos a los mencionados de Windows si bien tienen un nivel de desarrollo menor al haber menor demanda de los mismos. Aún así listaremos algunos proyectos. Muchos de ellos sólo funcionan para una arquitectura en concreto, habitualmente x86.

Linice: similar a Softice pero de código abierto. Ahora mismo está un poco abandonado y sólo funciona con kernels < 2.6.9 y 2.4.

<http://www.linice.com/>

Pice: private ice, similar a Linice.

<http://pice.sourceforge.net/>

RR0D: su objetivo es la portabilidad, pudiendo funcionar no sólo en Linux sino también en *BSD o Windows. Tiene una interfaz modo texto al contrario de softice y parecidos.

<http://rr0d.droids-corp.org/>

Fuentes y enlaces relacionados.

<http://www.linuxjournal.com/article/6100>

<http://www-128.ibm.com/developerworks/aix/library/au-unix-strace.html>

http://www.comp.glam.ac.uk/pages/staff/hggross/IntelligentTimingAnalysis_HTML/node20.html

<http://bulma.net/body.phtml?nIdNoticia=1805>

<http://pramode.net/articles/lfy/ptrace/pramode.html>

<http://linuxgazette.net/issue81/sandeep.html>