

# GNU Debugger (GDB)

“gdb” es un programa que permite depurar otros programas escritos en cualquier lenguaje de programación y compilados con el compilador gcc (*GNU Compiler Collection*, antes *GNU C Compiler*).

Depurar significa poder ejecutar el programa en un ambiente controlado de manera que permita controlar y estudiar el flujo de ejecución, la pila de llamadas, los valores de las variables, etcétera. Todo esto lo permite gdb.

Para ilustrar el uso de gdb, vamos a introducir un hipotético programa en C con un error de memoria, que suelen ser los más desastrosos.

## 1. El programa

Un aguerrido programador principiante en C ha escrito el siguiente programa para crear un buffer de cadenas de caracteres. Contiene dos errores. Uno que se podría considerar de escritura (el número que aparece en el for, al que se le ha añadido un cero de más), y otro de concepto, al no haber asignado memoria para cada uno de los punteros internos del buffer.

```
1  #include <stdio.h>
2
3  char** bufferCadenas;
4
5  char** inicializaBufferCadenas ()
6  {
7      char** tmp;
8      int i;
9
10     tmp = malloc(200*sizeof(char*));
11
12     for (i=0; i< 2000; i++)
13         tmp[i] = 0;
14
15     return tmp;
16 }
17
18 int
19 main(void)
20 {
21     char** buffer;
22
23     buffer = inicializaBufferCadenas();
24
25     strcpy(buffer[0], "hola");
26
27     printf("%s", buffer[0]);
28
29     return 0;
30 }
```

## 2. El indicio

Como una primera prueba, un intento de ejecución del programa nos da un resultado no esperado (imaginaremos que el programa se llama "pru"):

```
usuario@maquina:~$ ./pru
Violación de segmento (core dumped)
```

Esto suele ser un indicio bastante claro de que el programa no funciona. Este error indica que el programa ha accedido a una posición de memoria no asignada a sí mismo, lo que provoca una violación. Además, se nos crea un fichero adicional llamado "core".

## 3. El estudio

Para encontrar el o los problemas del código, utilizaremos gdb. Para que el depurador se pueda utilizar, se debe usar la opción "-g" en la compilación. Así, para generar nuestro programa "pru", deberemos escribir la siguiente línea:

```
usuario@maquina:~$ gcc -g -o pru pru.c
```

A continuación, para depurar, utilizamos el programa gdb de la siguiente manera:

```
usuario@maquina:~$ gdb ./pru
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-linux"...
(gdb)
```

Y como confirmación, podemos ejecutar el programa:

```
(gdb) r
Starting program: ./pru

Program received signal SIGSEGV, Segmentation fault.
0x0804848d in inicializaBufferCadenas () at pru.c:13
13             tmp[i] = 0;
```

Ahí vemos que el programa falla en la línea 13. Esto ya es una información bastante indicativa. Sin embargo, vayamos viendo las características de gdb haciendo una ejecución del programa paso a paso.

El primer concepto importante en gdb es la posibilidad de inclusión de puntos de ruptura (*breakpoints*). Podemos incluir uno en la función main de la siguiente forma:

```
(gdb) b main
Breakpoint 1 at 0x80484b2: file pru.c, line 23.
```

Si a continuación ejecutamos el programa, parará en la función especificada:

```
Starting program: ./pru
```

```
Breakpoint 1, main () at pru.c:23
23             buffer = inicializaBufferCadenas();
```

A continuación podemos decirle que el programa avance. Tenemos dos posibilidades: la orden "n", para que pase a la siguiente línea sin entrar en la llamada (aunque la llamada sí se ejecuta), o la orden "s", que se introduce dentro de la llamada a la función. Como sabemos que el código falla dentro de la función, entramos dentro con la orden "s":

```
(gdb) s
inicializaBufferCadenas () at pru.c:10
10             tmp = malloc(200*sizeof(char*));
```

Al pulsar "n", continuamos el programa:

```
(gdb) n
12             for (i=0; i< 2000; i++)
```

e incluso podemos imprimir el valor de tmp:

```
(gdb) print tmp
$1 = (char **) 0x8049698
```

y también listar el código en la posición donde estamos con la orden "l":

```
(gdb) l
7             char** tmp;
8             int i;
9
10            tmp = malloc(200*sizeof(char*));
11
12            for (i=0; i< 2000; i++)
13                tmp[i] = 0;
14
15            return tmp;
16        }
```

La línea que se va a ejecutar a continuación (12) se muestra en el centro del listado. Una inspección rápida nos dice que ha habido un error en el número dado en el bucle for. A continuación se verá cómo deducir este fallo de otra manera.

Para no tener este tipo de errores, es conveniente utilizar alguna constante o #define al principio del fichero:

```
#define BUFFER_LENGTH 200
```

## 4. Haciendo de Gil Grissom: depuración *post-mortem*

El programa contenía dos errores, como se vió. Al arreglar el fallo detectado anteriormente, y al volver a ejecutar el programa, fallará en la línea del strcpy:

```
Starting program: ./pru
```

```
Program received signal SIGSEGV, Segmentation fault.  
0x4008f90d in strcpy () from /lib/libc.so.6
```

Depurar los programas una vez han fallado resulta muy útil para identificar el punto exacto del error. Además, es lo que normalmente se hace, porque depurar paso a paso un programa hasta encontrar el error es algo muy tedioso. Una vez obtenido este mensaje, podemos pedirle la pila de llamadas con la orden "backtrace":

```
Program received signal SIGSEGV, Segmentation fault.  
0x4008f90d in strcpy () from /lib/libc.so.6  
(gdb) backtrace  
#0 0x4008f90d in strcpy () from /lib/libc.so.6  
#1 0x080484cf in main () at pru.c:25
```

Este listado indica que la función main se quedó ejecutando en la línea 25, y que después se llamó a la función de la librería libc "strcpy". La pila de llamadas guarda todas las funciones que han sido invocadas por el programa. En cualquier momento podemos seleccionar un "marco" de pila, lo que nos posicionaría la pila en la función que nos interese, pudiendo explorar las variables locales de esa función. En este caso nos interesa main. Para seleccionarla, utilizaremos la orden "frame":

```
(gdb) frame 1  
#1 0x080484cf in main () at pru.c:25  
25          strcpy(buffer[0], "hola");
```

Y luego podemos imprimir el valor de buffer[0], que ha sido el causante del error:

```
(gdb) print buffer[0]  
$1 = 0x0
```

Aquí está el error: buffer[0] es un puntero nulo. En la función inicializaBufferCadenas() se le debe reservar memoria (o hacerlo en la función main() ).

...oO\$Oo...

Otra forma de conseguir una depuración *post-mortem* es utilizando el fichero "core". Cuando el programa falla, produce este fichero que es un volcado de memoria del programa en ejecución:

```
-rw----- 1 usuario usuario 69632 2002-11-11 17:45 core  
-rwxr-xr-x 1 usuario usuario 14679 2002-11-11 17:05 pru
```

(si este fichero no es generado automáticamente, se puede obligar al shell a hacerlo escribiendo antes de ejecutar un programa: "ulimit -c 1024", para generar un fichero "core" de hasta 1MB).

Ejecutando gdb con la opción "--core", conseguimos dejar el programa justo en el punto de fallo:

```
usuario@maquina:~$ gdb --core=core
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
Core was generated by `./pru'.
Program terminated with signal 11, Segmentation fault.
#0  0x4008f90d in ?? ()
(gdb)
```

A continuación, con el comando "file" sirve para indicarle de qué fichero ejecutable debe leer los símbolos de depuración:

```
(gdb) file ./pru
Reading symbols from ./pru...done.
```

Y finalmente, podemos continuar la depuración *post-mortem* igual que antes:

```
(gdb) backtrace
#0  0x4008f90d in ?? ()
#1  0x080484cf in main () at pru.c:25
```

## 5. Cuadro resumen de órdenes de gdb

<b>Orden</b>	<b>Significado</b>
r [<argumentos>]	Ejecuta el programa desde el principio pasándole opcionalmente los argumentos como si le fueran dados en la línea de comandos.
print <expresión>	Imprime el valor de la expresión.
b <función> ó b <fichero>:línea	Establece un punto de ruptura.
c	Continúa la ejecución después de una interrupción.
s	Ejecuta la siguiente instrucción (entrando a las funciones llamadas).
n	Ejecuta la siguiente instrucción (sin entrar a las funciones llamadas).
l [<fichero>:línea] ó l [<función>]	Lista el código en el lugar especificado.
backtrace	Muestra la pila de llamadas en un momento dado (también <i>post-mortem</i> ).
frame <n>	Selecciona la función <i>i</i> -ésima de la pila de llamadas.
file <fichero>	Carga un fichero ejecutable para leer los símbolos que se utilizarán al depurar un fichero <i>core</i> .