



# **Control de corrupción y pérdida de memoria**

## Acerca de este documento

---

Muchas aplicaciones fallan de forma impredecible debido a que se corrompe la memoria, o simplemente acaban agotando toda la memoria del sistema tras un periodo prolongado de uso debido a pérdidas de memoria. Este reportaje explica cómo podemos detectar los problemas de corrupción y pérdida de memoria en nuestras aplicaciones usando una serie de herramientas que existen en Mac OS X para tal fin.

## Nota legal

---

Este reportaje ha sido escrito por Fernando López Hernández para MacProgramadores, y de acuerdo a los derechos que le concede la legislación española e internacional el autor prohíbe la publicación de este documento en cualquier otro servidor web, así como su venta, o difusión en cualquier otro medio sin autorización previa.

Sin embargo el autor anima a todos los servidores web a colocar enlaces a este documento. El autor también anima a cualquier persona interesada en conocer el lenguaje Objective-C a bajarse o imprimirse este reportaje.

Madrid, Febrero 2008

Para cualquier aclaración contacte con:

[fernando@DELITmacprogramadores.org](mailto:fernando@DELITmacprogramadores.org)

## Tabla de contenido

---

Corrupción de memoria.....	4
Pérdida de memoria .....	5
Las malloc tools.....	6
El comando leaks .....	10
El comando malloc_history .....	12
El comando heap.....	13
El comando mmap .....	15
Herramientas de productividad .....	18
Referencias.....	18

## Corrupción de memoria

---

La corrupción de memoria ocurre cuando el programa escribe datos en una zona de memoria distinta a la esperada. En este caso lo mejor que puede pasar es que el sistema operativo lo detecte y el programa casque. Si el sistema operativo no lo detecta, este cambio acabará afectando a otro trozo de programa que tarde o temprano fallará.

El error de corrupción de memoria más común en C es el del **desbordamiento de buffer**, el cual ocurre cuando un programa escribe más allá del trozo de memoria que teníamos reservada para él. Por ejemplo:

```
char* mi_copia = malloc (strlen(cadena));
strcpy (mi_copia, cadena);
```

Aquí estamos escribiendo un byte más allá del bloque de memoria reservado, para corregirlo deberíamos de haber reservado un byte más para que pudiéramos almacenar el carácter de final de cadena:

```
char* mi_copia = malloc (strlen(cadena)+1);
strcpy (mi_copia, cadena);
```

Este tipo de errores también es muy típico que se produzca por bucles que dan una vuelta más de las que deberían:

```
char digitos[10];
bool encontrado=false;
for (int i=0;i<=10;i++)
    digitos[i] = '0'+i;
```

Obsérvese que el bucle se repite 11 veces, y no 10. Lo peor de todo es que el bucle no falla, pero modificará la variable `encontrado` haciendo que luego el programa no se comporte como esperábamos.

Un desbordamiento en memoria dinámica reservada con `malloc()` puede provocar que el fallo se detecte mucho más tarde, ya que `malloc()` almacena información de contabilidad de los bloques asignados justo delante del puntero que nos devuelve, con lo que si sobrescribimos esta memoria el problema se producirá mucho más tarde, cuando ejecutemos los `free()` de esos bloques. Estos problemas pueden resultar extremadamente difíciles de detectar.

Otro efecto lateral del desbordamiento de búferes es que si sabemos que un programa no controla el tamaño de sus búferes, podemos enviar mucha información a este programa para que se desborde uno de sus búferes, y escribir así en la pila del programa, de forma que consigamos retornar a una de-

terminada dirección de memoria. Esta técnica la han utilizado en muchas ocasiones los hackers para burlar la seguridad de un sistema.

Otra forma de corrupción de memoria es la conocida como **puntero loco**, la cual se produce cuando la dirección de memoria a la que apunta un puntero no tiene relación con la dirección de memoria a la que realmente debería de apuntar. Un ejemplo se produce cuando a un trozo de memoria lo apuntamos con dos punteros y no actualizamos uno de ellos. Por ejemplo si tenemos el siguiente programa:

```
char* nombre_usuario;
const char* getNombreUsuario()
{
    return nombre_usuario;
}

void setNombreUsuario(const char* nombre)
{
    free(nombre_usuario);
    nombre_usuario = strdup(nombre); // Realiza un malloc()
}
```

Y ejecutamos las siguientes sentencias:

```
nombre = getNombreUsuario();
setNombreUsuario("Luis");
printf(nombre); //Aquí se está usando un puntero loco
```

En la última sentencia `nombre` apunta a una dirección de memoria liberada y en consecuencia `nombre` sería un puntero loco.

Si este documento le está resultando útil puede plantearse el ayudarnos a mejorarlo:

- Anotando los errores editoriales y problemas que encuentre y enviándolos al sistema de Bug Report de Mac Programadores.
- Realizando una donación a través de la web de Mac Programadores.

## **Pérdida de memoria**

---

La pérdida de memoria se produce cuando un programador hace un programa que reserva memoria, con `malloc()`, y olvida liberar esta memoria, con `free()`, cuando el programa ya no la va a usar más.

Este despiste normalmente no da problemas cuando el programa que vamos a ejecutar tiene una vida corta, ya que el sistema operativo libera toda la memoria reservada por un programa una vez que este termina (o falla).

En programas que pueden permanecer ejecutando durante semanas, meses o años (p.e. servidores) el problema se vuelve acumulativo, ya que cada vez que el programa pasa por un punto hace una reserva que nunca liberará. Esto explica porque Windows NT 4.0 sufría un proceso degenerativo tan rápido que le hacía ejecutar cada vez más lento y que transcurrida una semana había que reiniciarlo para que volviera a funcionar a un ritmo normal. Los servicios de Windows NT 4.0 padecían de este mal, y poco a poco se iban comiendo toda la memoria.

## Las malloc tools

Para detectar estos problemas, y otros muchos problemas de memoria, las librerías de reserva de memoria de Mac OS X proporcionan unas librerías de depuración. Aunque a estas librerías se las llama las **malloc tools**, en referencia a que es la función `malloc()` la que proporciona estas ayudas de depuración, en realidad se pueden usar con todos los sistemas de reserva de memoria de Mac OS X, como por ejemplo el operador `new` de C++, o el método de clase `alloc` en Objective-C

Para que estas librerías nos ayuden, lo único que tenemos que hacer es fijar determinadas variables de entorno antes de ejecutar una aplicación. Estas variables no es necesario fijarlas antes de compilar ya que es el propio runtime de la función `malloc()`, y no el compilador el que comprueba si estas variables de entorno están fijadas, y de hecho podemos depurar cualquier comando o programa de Mac OS X fijando estas variables. La Tabla 1 describe estas variables.

Variable de entorno	Descripción
<code>MallocHelp</code>	Muestra ayuda sobre cómo funcionan las malloc tools
<code>MallocLogFile file</code>	Permite indicar un fichero donde las malloc tools escriben los mensajes. Por defecto se escriben en <code>stderr</code> .
<code>MallocGuardEdges</code>	Añade dos páginas de guarda a los lados de cada bloque
<code>MallocDoNotProtectPrelude</code>	Quita la página de guarda anterior a los bloques (sólo es útil cuando usamos <code>MallocGuardEdges</code> )
<code>MallocDoNotProtectPostlude</code>	Quita la página de guarda posterior a los bloques (sólo es útil cuando usamos <code>MallocGuardEdges</code> )
<code>MallocScribble</code>	Ayuda a detectar lecturas de bloques liberados escribiendo <code>0x55</code> en todos los bytes al liberarlos.
<code>MallocStackLogging</code>	Almacena información de llamada a rutinas para el comando <code>malloc_history</code>

MallocStackLoggingNoCompact	Almacena información ampliada de llamada a rutinas para el comando malloc_history
-----------------------------	--

**Tabla 1:** Variables de entorno de las malloc tools

Vamos a comentar más detalladamente cómo funcionan estas variables de entorno.

## ***MallocHelp***

Esta variable de entorno sólo sirve para que las malloc tools nos muestren un mensaje de ayuda cada vez que vayamos a ejecutar un programa indicando que variables de entorno reconoce.

Para ver cómo funciona simplemente fije esta variable de entorno a cualquier valor. Por ejemplo, en bash haríamos:

```
$ export MallocHelp=si
```

Cualquier programa que ejecute ahora mostrará ayuda sobre las malloc tools:

```
$ ls
ls(589) malloc: environment variables that can be set for
debug:
- MallocLogFile <f> to create/append messages to file <f>
instead of stderr
- MallocGuardEdges to add 2 guard pages for each large block
- MallocDoNotProtectPrelude to disable protection (when
previous flag set)
- MallocDoNotProtectPostlude to disable protection (when
previous flag set)
- MallocStackLogging to record all stacks. Tools like leaks
can then be applied
- MallocStackLoggingNoCompact to record all stacks. Needed
for malloc_history
- MallocScribble to detect writing on free blocks and missing
initializers:
  0x55 is written upon free and 0xaa is written on allocation
- MallocCheckHeapStart <n> to start checking the heap after
<n> operations
- MallocCheckHeapEach <s> to repeat the checking of the heap
after <s> operations
- MallocCheckHeapSleep <t> to sleep <t> seconds on heap
corruption
- MallocCheckHeapAbort <b> to abort on heap corruption if <b>
is non-zero
- MallocErrorAbort to abort on a bad malloc or free
- MallocHelp - this help!
Compiladores IC Music Public Desktop Library bin Documents
Logica Pictures Sites tmp
```

Para poder depurar las aplicaciones gráficas con las malloc tools, éstas deberán de poder leer estas variables de entorno, para lo cual deberá ejecutarlas desde la consola, por ejemplo así:

```
$ open /Applications/Mozilla.app
```

Como ve, la variable de entorno `MallocHelp` sólo sirve para dar ayuda, vamos a comentar otras variables de entorno más interesantes.

### ***MallocGuardEdges, MallocDoNotProtectPrelude y MallocDoNotProtectPostlude***

`MallocGuardEdges` pone una página de memoria de 4KB sin permisos de acceso justo delante y detrás de cada bloque asignado. Esto permite capturar desbordamientos de buffer. La documentación indica que esto sólo se hace cuando se trata de bloques "grandes". Aunque no se especifica el valor de "grande", experimentalmente se puede comprobar que "grande" es cuando el bloque de memoria supera los 12KB.

```
/* mallocguard.m */

int main()
{
    char* memoria = (char*) malloc(1024*16);
    // Escribe fuera de la memoria reservada
    memoria[(1024*16)+1] = 'x';
    return 0;
}
```

**Listado 1:** Programa que escribe fuera de la memoria dinámica reservada

El Listado 1 muestra un programa que escribe fuera de la memoria dinámica reservada. Una vez compilado:

```
$ gcc mallocguard.m -o mallocguard
```

Si ejecutamos el programa sin fijar `MallocGuardEdges` no obtenemos ninguna indicación del problema:

```
$ ./mallocguard
```

Pero si fijamos esta variable de entorno:

```
$ export MallocGuardEdges=1
$ ./mallocguard
malloc[548]: protecting edges
Bus error
```

Nos detecta que nuestro ha escrito fuera del bloque permitido produciéndose un fallo de página.

Para acabar comentaremos que podemos quitar las páginas de guarda anterior o posterior al bloque de memoria reservado usando las opciones `MallocDoNotProtectPrelude` y `MallocDoNotProtectPostlude`.

## ***MallocScribble***

Esta opción hace que cuando se libere un bloque de memoria se escriba sobre todos sus bytes el valor 0x55, lo cual sirve para capturar intentos de utilizar ese bloque después de haber sido liberado. Obsérvese que 0x55 se ha elegido porque es un valor impar y cualquier intento de dereferenciar un puntero que apunte a un valor impar produce una excepción (odd address), con lo que si en la estructura de memoria liberada hubiera alguna variable de tipo puntero que luego quisiéramos utilizar se va a producir esta excepción. Por desgracia, experimentalmente hemos comprobado que `free()` siempre deja los 8 primeros bytes a 0, lo cual puede dificultar la captura de errores.

```
/* mallocscribble.m */
#import <stdlib.h>

typedef struct
{
    char nombre[20];
    char apellidos[30];
} Persona;

int main()
{
    Persona* p = malloc(sizeof(Persona));
    strcpy(p->nombre, "Fernando");
    strcpy(p->apellidos, "Lopez");
    printf("Nombre:%s Apellidos:%s\n", p->nombre, p->apellidos);
    free(p);
    printf("Nombre:%s Apellidos:%s\n", p->nombre, p->apellidos);
    return 0;
}
```

**Listado 2:** Programa que escribe en memoria liberada

El Listado 2 muestra un programa que escribe en memoria liberada por `free()`. Al ejecutarlo obtenemos:

```
$ ./mallocscribble
Nombre:Fernando Apellidos:Lopez
Nombre: Apellidos:Lopez
```



```

.....
while(1);
return 0;
}

```

Por ejemplo, imaginemos que tenemos el programa del Listado 3, este programa pierde memoria cada vez que asignamos un nuevo valor a `bloque1`.

```

/* pierde.c */
#include <stdio.h>
#include <stdlib.h>

main()
{
    char* bloque1 = (char*)malloc(1024);
    printf("Perder memoria (s/n)?");
    char c;
    while( (c=getchar()) == 's' )
    {
        bloque1= (char*)malloc(1024);
        getchar(); // Quita el '\n'
        printf("Perder memoria (s/n)?");
    }
    return 0;
}

```

**Listado 3:** Programa que pierde memoria

Para ver cómo el comando `leaks` detecta las pérdida de memoria, primero debemos de ejecutar este programa en una consola. Conviene exportar la variable de entorno `MallocStackLogging`, ya que si esta variable de entorno existe al lanzar el proceso, `leaks` muestra también la pila de llamadas a funciones:

```

$ gcc pierde.c -o pierde
$ export MallocStackLogging=1
$ ./pierde
pierde(328) malloc: recording malloc stacks to disk using
standard recorder
pierde(328) malloc: stack logs being written into /tmp/stack-
logs.328.pierde.UNV6o2
Perder memoria (s/n)?

```

En los logs de `malloc()` se nos informa de que el número de proceso es 328, y usamos este número de proceso para lanzar `leaks` en otra consola<sup>1</sup>:

```

$ leaks 328
Process 328: 10 nodes malloced for 10 KB
Process 328: 0 leaks for 0 total leaked bytes.

```

<sup>1</sup> En este caso, como sólo existe un proceso con el nombre `pierde`, también podríamos haber pasado como argumento el nombre del proceso.

Vemos que inicialmente no hay ninguna pérdida de memoria detectada, pero si ahora hacemos que `pierde` pierda memoria:

```
Perder memoria (s/n)?s
Perder memoria (s/n)?
```

Y volvemos a ejecutar `leaks`:

```
$ leaks 328
Process 328: 11 nodes malloced for 11 KB
Process 328: 1 leak for 1024 total leaked bytes.
Leak: 0x801000 size=1024
0x00000000 0x00000000 0x00000000 0x00000000 .....
0x00000000 0x00000000 0x00000000 0x00000000 .....
0x00000000 0x00000000 0x00000000 0x00000000 .....
0x00000000 0x00000000 0x00000000 0x00000000 .....
0x00000000 0x00000000 0x00000000 0x00000000 .....
0x00000000 0x00000000 0x00000000 0x00000000 .....
0x00000000 0x00000000 0x00000000 0x00000000 .....
0x00000000 0x00000000 0x00000000 0x00000000 .....
0x00000000 0x00000000 0x00000000 0x00000000 .....
...
Call stack: [thread 0xa007d074]: | 0x0 | start | main | malloc
| malloc_zone_malloc
```

Vemos que el comando `leak` acaba de detectar la pérdida de memoria en la llamada a `malloc()` dentro de la función `main()`.

## El comando `malloc_history`

---

`malloc_history` nos permite inspeccionar un proceso en ejecución y ver las asignaciones de memoria que ha hecho.

Para que este comando funcione necesitamos exportar la variable de entorno `MallocStackLogging`. Por ejemplo, podemos depurar nuestro anterior programa así:

```
$ export MallocStackLogging=1
$ ./pierde
pierde(380) malloc: recording malloc stacks to disk using
standard recorder
pierde(380) malloc: stack logs being written into /tmp/stack-
logs.380.pierde.Hqe7Ze
Perder memoria (s/n)?s
Perder memoria (s/n)?
```

Ahora desde otra consola ejecutamos:

```
$ $malloc_history 380 -all_by_size
```

```
1 calls for 4096 bytes: thread_a007d074 |0x0 | start | main |
getchar | __srget | __srefill | __smakebuf | malloc |
malloc_zone_malloc
```

```
1 calls for 4096 bytes: thread_a007d074 |0x0 | start | main |
printf$LDBL128 | vfprintf_l$LDBL128 | __vfprintf$LDBL128 |
__swsetup | __smakebuf | malloc | malloc_zone_malloc
```

```
1 calls for 1024 bytes: thread_a007d074 |0x0 | start | main |
malloc | malloc_zone_malloc
```

```
1 calls for 1024 bytes: thread_a007d074 |0x0 | start | main |
malloc | malloc_zone_malloc
```

```
1 calls for 64 bytes: thread_a007d074 |0x0 | start | main |
printf$LDBL128 | vfprintf_l$LDBL128 | __vfprintf$LDBL128 |
localeconv_1 | malloc | malloc_zone_malloc
```

Aquí aparecen todas las llamadas a `malloc()` que ha hecho nuestro proceso (directamente o a través de subrutinas).

Una utilidad de este comando es una zona detectar accesos a bloques de memoria liberada, combinándolo con la variable de entorno `MallocScribble`, la cual escribía `0x55` en los búferes liberados. Si hemos intentado escribir en un buffer liberado, `malloc_history` mostrará un mensaje de advertencia avisándonos de tal hecho.

## El comando heap

---

El comando `heap` muestra todos los objetos creados en el heap de un proceso. El heap puede estar formado por una o más zonas. Las aplicaciones C suelen tener sólo una zona llamada **zona de memoria estándar** o **zona por defecto**. Las aplicaciones Objective-C pueden tener zonas de memoria adicionales donde se almacenan los objetos Objective-C.

En el apartado anterior podemos ver que la función `malloc()` siempre llama a la función `malloc_zone_malloc()` que se encarga de decidir la zona de memoria donde reservar la memoria.

Por ejemplo, si preguntamos por las zonas de memoria de Bash (pasando como argumento el identificador de proceso o bien el nombre del proceso) obtenemos una sola zona (`DefaultMallocZone_0xa7000`):

```
$ heap bash
Process 196: 1 zone
Zone DefaultMallocZone_0xa7000: Overall size: 9295KB; 2552
nodes malloced for 194KB (2% of capacity); largest unused:
[0x810600-8126KB]
```

```
Zone DefaultMallocZone_0xa7000: 2552 nodes - Sizes: 80KB[1]
4KB[1] 2560[14] 2048[2] 1024[9] 512[2] 496[1] 432[6] 384[2]
352[1] 256[6] 240[4] 224[2] 176[1] 160[2] 144[2] 112[8] 96[10]
80[38] 64[26] 48[44] 32[438] 16[1932]
```

Can't get the ObjC classes for process 196

Can't get the CTypes for process 196

```
-----
Zone DefaultMallocZone_0xa7000: 2552 nodes (197728 bytes)
```

CLASS_NAME	TYPE	BINARY	COUNT	BYTES	AVG
=====	=====	=====	=====	=====	=====
<non-object>			2552	197728	77.5

El comando `heap` es capaz de identificar el tipo de los objetos en el heap (Objective-C, C++, CType). Por ejemplo si preguntamos por el heap de Finder obtenemos dos zonas de memoria: `DefaultMallocZone_0x51a000` con la memoria reservada por `malloc()` y `unnamed_zone_0x51d000` con la memoria reservada por Objective-C. Además obtenemos un listado de los objetos creados en el heap y su tipo

#### \$ heap Finder

Process 154: 2 zones

Zone DefaultMallocZone\_0x51a000: Overall size: 11027KB; 21798 nodes malloced for 2289KB (20% of capacity); largest unused: [0x88d000-7627KB]

Zone unnamed\_zone\_0x51d000: Overall size: 1023KB; 2 nodes malloced for 1KB (0% of capacity); largest unused: [0x1000070-1023KB]

All zones: 21800 nodes malloced - 2289KB

```
Zone DefaultMallocZone_0x51a000: 21798 nodes - Sizes: 236KB[1]
112KB[1] 60KB[1] 56KB[1] 48KB[1] 36KB[1] 28KB[1] 24KB[7]
20KB[1] 16KB[1] 12KB[1] 11KB[1] 9KB[1] 9KB[1] 8KB[6] 7KB[2]
6KB[3] 6KB[3] 5KB[1] 4KB[16] 3584[4] 2560[7] 2048[26] 1536[25]
1024[134] 512[62] 496[1] 480[5] 448[8] 432[6] 416[3] 400[6]
384[18] 368[10] 352[13] 336[9] 320[25] 304[9] 288[15] 272[50]
256[134] 240[42] 224[72] 208[21] 192[259] 176[66] 160[127]
144[249] 128[349] 112[238] 96[351] 80[839] 64[3119] 48[2478]
32[5950] 16[7017]
```

```
Zone unnamed_zone_0x51d000: 2 nodes - Sizes: 64[1] 48[1]
```

```
All zones: 21800 nodes malloced - Sizes: 236KB[1] 112KB[1]
60KB[1] 56KB[1] 48KB[1] 36KB[1] 28KB[1] 24KB[7] 20KB[1]
16KB[1] 12KB[1] 11KB[1] 9KB[1] 9KB[1] 8KB[6] 7KB[2] 6KB[3]
6KB[3] 5KB[1] 4KB[16] 3584[4] 2560[7] 2048[26] 1536[25]
1024[134] 512[62] 496[1] 480[5] 448[8] 432[6] 416[3] 400[6]
384[18] 368[10] 352[13] 336[9] 320[25] 304[9] 288[15] 272[50]
256[134] 240[42] 224[72] 208[21] 192[259] 176[66] 160[127]
144[249] 128[349] 112[238] 96[351] 80[839] 64[3120] 48[2479]
32[5950] 16[7017]
```

Found 3822 ObjC classes in process 154  
 Found 112 CTypes in process 154

CLASS_NAME	TYPE	BINARY	COUNT	BYTES	AVG
=====	=====	=====	=====	=====	=====
<non-object>			11758	1831616	155.8
NSCFString	ObjC	CoreFoundation	5422	249216	46.0
NSCFArray	ObjC	Foundation	631	22016	34.9
TPropertyInfo	C++	DesktopServices	324	10368	32.0
NSURL	ObjC	Foundation	105	3360	32.0
TSMInputSource	CType	HIToolbox	1	16	16.0
calantlr::Token	C++	CalendarStore	1	16	16.0
.....					

## El comando vmmap

El comando `vmmap` nos permite ver las regiones de la memoria de un proceso. Como en los casos anteriores, el proceso se puede indicar o bien por nombre o bien por número de proceso. Por ejemplo:

```
$ vmmap -submap bash
Virtual Memory Map of process 176 (bash)
Output report format: 2.2 -- 32-bit process

==== Non-writable regions for process 176
__PAGEZERO          00000000-00001000 [   4K] ---/--- SM=NUL
                    /bin/bash
__TEXT              00001000-00098000 [ 604K] r-x/rwx SM=COW
                    /bin/bash
__LINKEDIT          000a2000-000a6000 [   16K] r--/rwx SM=COW
                    /bin/bash
STACK GUARD         000a6000-000a7000 [   4K] ---/rwx SM=NUL
STACK GUARD         000a8000-000a9000 [   4K] ---/rwx SM=NUL
__TEXT              8fe00000-8fe31000 [ 196K] r-x/rwx SM=COW
                    /usr/lib/dyld
__LINKEDIT          8fe68000-8fe76000 [   56K] r--/rwx SM=COW
                    /usr/lib/dyld
Submap              90000000-a0000000          r--/rwx
                    machine-wide submap
__TEXT              9033e000-904d8000 [ 1640K] r-x/r-x SM=COW
                    /usr/lib/libSystem.B.dylib
__TEXT              90dd5000-90ecc000 [   988K] r-x/r-x SM=CO
                    /usr/lib/libiconv.2.dylib
__TEXT              9375d000-9378f000 [   200K] r-x/r-x SM=COW
                    /usr/lib/libncurses.5.4.dylib
__TEXT              95897000-9589d000 [   24K] r-x/r-x SM=CO
                    /usr/lib/system/libmathCommon.A.dylib
__TEXT              958ac000-958b8000 [   48K] r-x/r-x SM=COW
                    /usr/lib/libgcc_s.1.dylib
__LINKEDIT          96e6e000-972da000 [ 4528K] r--/r-- SM=COW
                    /usr/lib/system/libmathCommon.A.dylib
Submap              a0d17000-b0000000          r--/rwx
                    process-only submap
STACK GUARD         bc000000-bf800000 [ 56.0M] ---/rwx SM=NUL
```

```

STACK GUARD          fffec000-ffff0000 [ 16K] ---/rwx SM=NUL
Submap              ffff8000-fffff000
                    process-only submap
shared memory       ffff8000-ffffa000 [  8K] r-x/r-x SM=SHM

==== Writable regions for process 176
__DATA              00098000-000a2000 [  40K] rw-/rwx SM=COW
                    /bin/bash
MALLOC (freed?)     000a7000-000a8000 [   4K] rw-/rwx SM=PRV
MALLOC_LARGE        000a9000-000bd000 [  80K] rw-/rwx SM=COW
                    DefaultMallocZone_0x100000
MALLOC_TINY         00100000-00200000 [ 1024K] rw-/rwx SM=COW
                    DefaultMallocZone_0x100000
MALLOC_SMALL        00800000-01000000 [ 8192K] rw-/rwx SM=COW
                    DefaultMallocZone_0x100000
__DATA              8fe31000-8fe68000 [ 220K] rw-/rwx SM=COW
                    /usr/lib/dyld
Submap              90000000-a0000000
                    machine-wide submap
__DATA              a0088000-a0106000 [ 504K] rw-/rwx SM=COW
                    /usr/lib/libSystem.B.dylib
__DATA              a016c000-a016d000 [   4K] rw-/rwx SM=COW
                    /usr/lib/libiconv.2.dylib
__DATA              a058f000-a0598000 [  36K] rw-/rwx SM=COW
                    /usr/lib/libncurses.5.4.dylib
__DATA              a0a0d000-a0a0e000 [   4K] rw-/rwx SM=COW
                    /usr/lib/system/libmathCommon.A.dylib
__DATA              a0a28000-a0a29000 [   4K] rw-/rwx SM=COW
                    /usr/lib/libgcc_s.1.dylib
Submap              a0d17000-b0000000
                    process-only submap
Stack               bf800000-bffff000 [ 8188K] rw-/rwx SM=COW
Stack               bffff000-c0000000 [   4K] rw-/rwx SM=PRV
                    thread 0
Submap              ffff8000-fffff000
                    process-only submap

```

==== Legend

SM=sharing mode:

COW=copy\_on\_write PRV=private NUL=empty ALI=aliased

SHM=shared ZER=zero\_filled S/A=shared\_alias

==== Summary for process 176

ReadOnly portion of Libraries: Total=8300K resident=7084K(85%)

swapped\_out\_or\_unallocated=1216K(15%)

Writable regions: Total=17.3M written=88K(0%)

resident=324K(2%) swapped\_out=0K(0%) unallocated=17.0M(98%)

```

REGION TYPE          [ VIRTUAL]
=====
MALLOC                [  9300K]
STACK GUARD          [  56.0M]
Stack                [  8192K]
__DATA                [   812K]
__LINKEDIT           [  4600K]
__PAGEZERO           [    4K]

```

```

__TEXT          [  3700K]
shared memory   [    8K]

```

El comando `vmmap` muestra para cada región la dirección de memoria de inicio y final, su tamaño, los permisos, el modo de compartición y la imagen de la región. Las regiones que muestra `vmmap` son tanto las regiones de la imagen del programa como las regiones de las imágenes de las librerías con las que está enlazado el programa.

Es importante tener en cuenta que las regiones pueden tener páginas no reservadas con lo que el tamaño de las regiones será mayor a la memoria realmente consumida por el proceso. Por ejemplo, un fichero mapeado en memoria representaría una región que no está cargada en memoria hasta que no se lea o escriba en todas las páginas de la región. En el ejemplo de ejecución anterior la región `STACK GUARD` es especialmente grande, pero seguramente no todas sus páginas están cargadas en memoria. La razón por la que la región de guarda de pila suele ser especialmente grande es para permitir que la pila del proceso pudiera crecer sin producirse un desbordamiento de pila.

Es importante no confundir las *regiones*, que son agrupaciones de páginas en que se divide la memoria de un proceso con las *zonas*, que son agrupaciones de páginas en el heap.

Los **permisos** de cada región describen si en una región se puede leer, escribir y ejecutar su contenido. Cada región tiene asociados unos permisos actuales (que se muestran primero) y unos permisos máximos (que se muestran a continuación). Por ejemplo, la región `__PAGEZERO`, que consta de una sola página y empieza siempre en la dirección `0x00000000`, no permite lecturas, escrituras, ni ejecución (---). De esta forma cualquier intento de acceso a la dirección de memoria `NULL` produce un error de bus. Las páginas de los ejecutables (p.e. la región `__TEXT`) tienen siempre los permisos actuales `r-x` que permiten leer y ejecutar su contenido, pero no modificarlo. Sin embargo sus permisos máximos permiten escribir en la región, con lo que estas regiones suelen tener la máscara de permisos `r-x/rwx`.

El **modo de compartición** indica si la página es compartida con otros procesos y qué pasa cuando la página se modifica. Las regiones privadas (`PRV`) – como por ejemplo la pila del proceso – son regiones sólo visibles para el proceso. Las regiones aliased (`ALI`) y shared (`SHM`) son regiones compartidas por todos los procesos. Estas regiones suelen proceder de librerías de enlace dinámico y normalmente sólo tienen permisos de lectura y ejecución. Las regiones Copy On write (`COW`) son regiones compartidas por muchos procesos pero que cuando un proceso escribe sobre ellas se hace una copia de sus páginas para el proceso y se convierten en regiones privadas. Actualmente las regiones `COW` no cambian su tipo a `PRV` hasta que todas sus páginas han sido escritas por el proceso. Las regiones Zero filled (`ZER`) son regiones inicializa-

das con ceros que se suelen usar para almacenar las variables globales del proceso.

Si ejecutamos `vmmap` con la opción `-submap` (como en el ejemplo de ejecución anterior) la salida de `vmmap` incluye una descripción de los submapas. Un **submapa** es un conjunto de páginas que el sistema operativo reutiliza entre muchos procesos. De esta forma se reduce considerablemente el consumo de memoria. Por ejemplo, la memoria entre `0x90000000` y `0x9fffffff` se usa por librerías de uso muy común como son las librerías `libSystem.B.dylib` o `libgcc_s.1.dylib`. Los submapas pueden ser, o bien compartidos por todos los procesos (machine-wide) o locales al proceso (process-only).

El submapa machine-wide situado en la región entre `0x90000000` y `0x9fffffff` contiene todas las librerías de uso común, y en la documentación muchas veces se le llama **split library region**. Aquí se encuentran siempre cargadas un gran número de librerías de uso común. Por defecto el comando `vmmap` sólo muestra las regiones de la split library region que están siendo usadas por el proceso, si queremos que se muestren todas las regiones debemos de pasar al comando `vmmap` la opción `-allSplitLibs`.

## Herramientas de productividad

---

Mac OS X dispone de dos herramientas llamadas `Instruments.app` y `MallocDebug.app` que mejoran la productividad del programador al poder realizar el análisis de memoria de forma gráfica. Podemos encontrar estas herramientas en el directorio `/Developer/Applications`. Recomendamos al lector que las eche un vistazo.

## Referencias

---

Para obtener más información de estos comandos puede visitar:

Memory corruption and malloc tools

[http://old.macedition.com/bolts/bolts\\_20021210.php](http://old.macedition.com/bolts/bolts_20021210.php)

What are MemoryLeaks?

<http://www.cocoadev.com/index.pl?MemoryLeaks>

OS X heap exploitation techniques

[http://felinemenace.org/papers/p63-0x05\\_OSX\\_Heap\\_Exploitation\\_Techniques.txt](http://felinemenace.org/papers/p63-0x05_OSX_Heap_Exploitation_Techniques.txt)