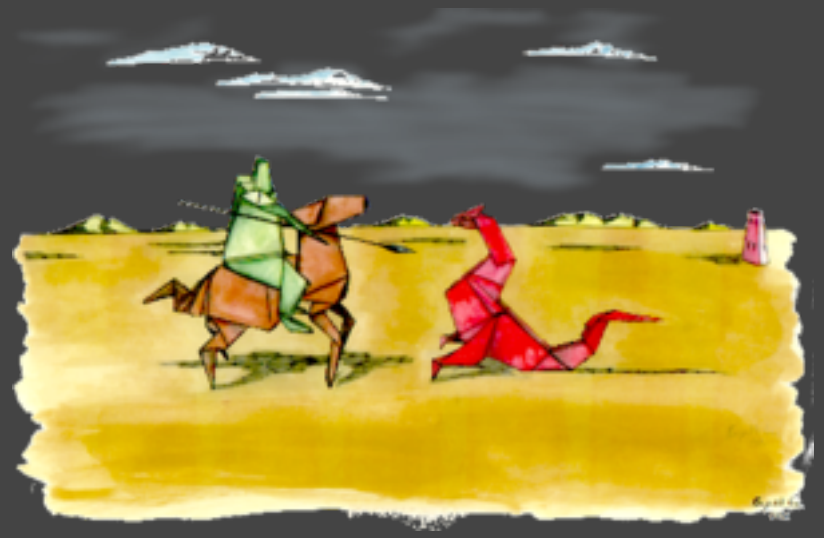

# Valgrind

Tutorial basico de Memcheck

# ¿Que es Valgrind?

Valgrind es un sistema para debuggear y hacer profiling (investigar el comportamiento de un programa utilizando información obtenida durante su ejecución) sobre programas en Linux. Es de licencia GPL.

- Una de las principales características de Valgrind es poder detectar automáticamente muchos errores de manejo de memoria, (como por ejemplo memory leaks) el objetivo es lograr programas mas estables.
- Una de las principales ventajas es que Valgrind funciona con programas escritos en cualquier lenguaje. Al trabajar directamente sobre los binarios, Valgrind puede analizar programas compilados y programados en cualquier lenguaje de programación, sean compilados, pre-compilados o interpretados.

Valgrind esta pensado para funcionar principalmente con programas escritos en C y C++, ya que estos son los lenguajes que introducen este tipo de bugs con mayor facilidad. Pero puede ser usado, por ejemplo, en programas escritos incluso en varios lenguajes diferentes (C, C++, Java, Perl, Python, assembly, Fortran, y muchos otros).

# Memcheck

Herramienta que detecta problemas de manejo de memoria. Cuando un programa corre bajo Memcheck, se verifican todas sus lecturas y escrituras en memoria, y se interceptan todas las llamadas a malloc/new/free/delete. Esto le permite a Memcheck detectar si el programa:

- Accede a memoria a la cual no deberia (areas no reservadas, areas que ya fueron liberadas, areas mas alla del final de heap blocks, partes inaccesibles del stack).
- Usa valores no inicializados de manera riesgosa.
- Pierde memoria (tiene memory leaks).
- Libera memoria incorrecta (double frees).

Memcheck reporta estos errores en el momento en que ocurren, indicando la linea del codigo fuente en el cual se generan, asi como tambien la traza de llamadas a funciones mediante la cual el programa llego a esa linea.
Memcheck corre programas aproximadamente 10 a 30 veces mas **lento** que lo normal.

# Ejecutando Memcheck

Un programa simple:

```
#include <stdlib.h>
void funcion(void){
    int * x = malloc(10 * sizeof(int));
    x[10] = 0;
}
int main(void){
    funcion();
    return 0;
}
```

¿Errores?

# Ejecutando Memcheck

Un programa simple:

```
#include <stdlib.h>
void funcion(void){
    int * x = malloc(10 * sizeof(int));
    x[10] = 0;
}
int main(void){
    funcion();
    return 0;
}
```

¿Errores?

```
#include <stdlib.h>
void funcion(void){
    int * x = malloc(10 * sizeof(int));
    x[10] = 0;        // problema 1: heap block overrun
}                     // problema 2: memory leak -- x no liberado
int main(void){
    funcion();
    return 0;
}
```

# Ejecutando Memcheck

Al igual que al utilizar **gdb**, compilamos nuestro programa con *-g*, esto incluye información de debugging en nuestro ejecutable, que Memcheck va a utilizar para obtener los números de linea de los errores.

Ejecución común:

```
~ ./programa
```

Ejecución con Valgrind:

```
~ valgrind --tool=<herramienta> <argumentos> programa
```

*Memcheck* es la herramienta default, por lo que no es necesario utilizar --tool

Ademas, queremos que Memcheck nos de la mayor información posible sobre nuestros leaks, para lograr esto utilizamos *--leak-check=yes*, que retorna la mayor cantidad de detalles posibles.

```
~ valgrind --leak-check=yes programa
```

# Interpretando el resultado

Primer problema, *heap block overrun*: se da cuando escribimos en un lugar de la memoria no permitido

```
==19182== Invalid write of size 4
==19182==    at 0x804838F: funcion (ejemplo.c:4)
==19182==    by 0x80483AB: main (ejemplo.c:7)
==19182== Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: funcion (ejemplo.c:3)
==19182==    by 0x80483AB: main (ejemplo.c:7)
```

# Interpretando el resultado

Primer problema, *heap block overrun*: se da cuando escribimos en un lugar de la memoria no permitido

Tipo de error

Stack trace

```
==19182== Invalid write of size 4
==19182==    at 0x804838F: funcion (ejemplo.c:4)
==19182==    by 0x80483AB: main (ejemplo.c:7)
==19182== Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: funcion (ejemplo.c:3)
==19182==    by 0x80483AB: main (ejemplo.c:7)
```

Ubicación de la memoria involucrada

- Tipo de error: "invalid write" nos indica que estamos tratando de escribir en una porción de memoria que no hemos reservado, en este caso, al tratar de escribir mas allá del tamaño del arreglo.
- Stack trace: nos indica lo que estaba en el stack al momento del error, es mas cómodo leerlo de abajo hacia arriba, y nos da el camino que siguió la ejecución hasta la escritura inválida.
- Ubicación de la memoria: Indica el lugar de la memoria en el cual se intento escribir inválidamente.

# Interpretando el resultado

Segundo problema, *memory leak*: No liberamos memoria reservada

```
==19182== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: funcion (ejemplo.c:3)
==19182==    by 0x80483AB: main (ejemplo.c:7)
```

# Interpretando el resultado

Segundo problema, *memory leak*: No liberamos memoria reservada

Tipo de error

Stack trace

```
==19182== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: funcion (ejemplo.c:3)
==19182==    by 0x80483AB: main (ejemplo.c:7)
```

- Tipo de error: nos indica que una cantidad determinada de bytes se perdieron, ya sea porque la ejecución terminó y no se liberaron o porque perdimos referencia a su ubicación.
- Stack trace: nos indica el momento preciso en el cual fue reservada la memoria que se perdió.

# Interpretando el resultado

Para tener en cuenta:

- Evitar dobles free: Por regla general, siempre debo liberar la memoria que pedí explícitamente. Nunca debo liberar memoria pedida por C (como un arreglo).

- La causa mas común para la escritura/lectura inválida es tratar de acceder a un arreglo con un índice erróneo, esto se da generalmente al recorrer todos los elementos de un arreglo/lista con un ciclo. Hay que tener siempre claras las condiciones de salida del ciclo (y recordar que los arreglos comienzan en cero!)

- Ser prolijo: tratar siempre de mantener el control de la memoria en el nivel de abstracción que corresponda. Es decir: si un TAD pide memoria al ser creado, tiene que liberar esa memoria al ser destruido; Si pido memoria para manejar las entradas del usuario en la interfaz, debería liberarla cuando ya no la utilice, etc...
- Conocer mis funciones: ¿Retorno copias de la información o referencias? tener en claro que las funciones que devuelven copias de la información están pidiendo memoria que luego debe ser liberada: utilizar una función de este tipo anidada, (por ejemplo dentro de una condición de un **if** o un **while**) generalmente significa perder la referencia a esa memoria y generar un leak.

# Bibliografia

**Bibliografia**


- http://valgrind.org/
- http://valgrind.org/docs/manual/dist.install.html
- http://en.wikipedia.org/wiki/Valgrind