Capítulo 17. El perfilador de memoria Valgrind

Tabla de contenidos

Valgrind es un sistema de depurado y perfilado para programas Linux, de utilidad a la hora de detectar muchos problemas de gestión de memoria, errores en el uso de hilos, que hace que los programas sean más estables. Desafortunadamente no puede resolver todos los problemas en un trozo de código, por lo que requiere de iteración con el programador, el cual debe de analizar la salida de este programa. En este documento, se introduce de forma práctica dos de las herramientas de Valgrind: Memcheck y Helgrind.

17.1. Herramienta Memcheck

Hay varias herramientas empaquetadas con Valgrind. La por defecto y quizás la más usada es Memcheck. Esta herramienta inserta código adicional alrededor de muchas instrucciones para comprobar su validez y alcanzabilidad. Memcheck reemplaza el algoritmo de gestión de memoria que viene en C por uno propio, que incluye guardas alrededor de todos los bloques de memoria reservados. El tipo de problemas que detecta incluye principalmente: (1) uso de memoria sin inicializar, (2) lectura/escritura de memoría después de que se libere, (3) lectura/escritura en secciones ilegales, y (4) potenciales fugas de memoria.

Antes de entrar en los diferentes tipos de problemas que detecta la herramienta, se enseña como interactuar con la herramienta, mediante un ejemplo no presenta ninguna anomalía, pero que ilustrará la funcionalidad de la herramienta. Se trata de un programa hola-mundo que que no realiza ningún tipo de operación.

```
1 // Save this code as valgrind_hello_good.c
2 // Compile with compiler:   $ gcc -Wall -gstabs valgrind_hello_good.c -o valgrind_hello_good
3 // Test with  valgrind:     $ valgrind --tool=memcheck  ./valgrind_hello_good
4 //
5 #include <stdlib.h>
6 int main()
7 {
8   return 0;
9 }
```

Este ejemplo sencillo no presenta anomalías detectables por Memcheck. Tal y como se verá ahora, tras la compilación del código fuente: `valgrind_hello_good.c` con `gcc` y su ejecución de la imagen creada por el gcc mediante Valgrind-Memcheck. La traza de ejecución muestra cero errores:

```
$ gcc -Wall -gstabs valgrind_hello_good.c -o valgrind_hello_good
```

```
$ valgrind --tool=memcheck  --leak-check=full -v ./valgrind_hello_good

==17624== Memcheck, a memory error detector
==17624== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==17624== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==17624== Command: ./valgrind_hello_good

==17635== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==17635== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Siguiendo con el ejemplo, le introducimos una fuga de memoria (memory leak) mediante un `malloc` en el código del `main`:

```
1  // Save as valgrind_hello_bad.c
2  // Compile with debug info: $ gcc  -Wall -gstabs valgrind_hello_bad.c -o valgrind_hello_bad
3  // Test with  valgrind:     $ valgrind --tool=memcheck  ./valgrind_hello_bad
4  //
5  #include <stdlib.h>
6  int main()
7  {
8    void* ptr=malloc(1);
9    return 0;
10 }
```

Ahora Valgrind nos muestra que ha habido una fuga de memoria, en su salida:

```
$ gcc -Wall -gstabs valgrind_hello_bad.c -o valgrind_hello_bad
$ valgrind --tool=memcheck  --leak-check=full -v ./valgrind_hello_bad

==17722== HEAP SUMMARY:
==17722==     in use at exit: 1 bytes in 1 blocks
==17722==   total heap usage: 1 allocs, 0 frees, 1 bytes allocated
==17722==
==17722== Searching for pointers to 1 not-freed blocks
==17722== Checked 55,996 bytes
==17722==
==17722== 1 bytes in 1 blocks are definitely lost in loss record 1 of 1
==17722==    at 0x402BE68: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==17722==    by 0x80483F8: main (valgrind_hello_bad.c:8)
==17722==
==17722== LEAK SUMMARY:
==17722==    definitely lost: 1 bytes in 1 blocks
==17722==    indirectly lost: 0 bytes in 0 blocks
==17722==      possibly lost: 0 bytes in 0 blocks
==17722==    still reachable: 0 bytes in 0 blocks
==17722==         suppressed: 0 bytes in 0 blocks
==17722==
==17722== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
==17722== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Si queremos mantener el `malloc`, una forma de atajar la fuga de memoria es con un `free` justo antes de que finalice el código del `main`:

```
1  // Save as valgrind_hello_bad_solved.c
2  // Compile with debug info: $ gcc -gstabs valgrind_hello_bad_solved.c -o valgrind_hello_bad_solved
3  // Test with  valgrind:     $ valgrind --tool=memcheck  ./valgrind_hello_bad_solved
4  //
5  #include <stdlib.h>
6  int main()
7  {
8    void* ptr=malloc(1);
```

```
 9   free(ptr);
10   return 0;
11 }
```

17.1.1. Lecturas/escrituras ilegales

Uno de los problemas sobre los que advierte es el código es sobre la lectura/escritura ilegal por parte del programa. Veamos con un ejemplo de escritura ilegal sobre la posición de memoria cero. Esta escritura ilegal se hace accediendo a una posición de memoria que no pertenece al programa y que lanza una alarma en Memcheck:

```
1  // Illegal write
2  // gcc -gstabs valgrind_illegal_write.c -o valgrind_illegal_write
3  // $ valgrind  -v ./valgrind_illegal_write
4  #include <stdlib.h>
5  int main()
6  {
7    int *ptr=0;
8    (*ptr)=33;
9    return 0;
10 }
```

Vemos como Valgrind lo detecta (con escritura ilegal) en la función `main`:

```
$ gcc -gstabs -Wall valgrind_illegal_read_write.c -o valgrind_illegal_read_write
$ valgrind -v ./valgrind_illegal_read_write

--6429--   .. CRC is valid
--6429-- REDIR: 0x40d01e0 (strnlen) redirected to 0x40254a0 (_vgnU_ifunc_wrapper)
--6429-- REDIR: 0x40d1730 (strncasecmp) redirected to 0x40254a0 (_vgnU_ifunc_wrapper)
--6429-- REDIR: 0x40d0380 (__GI_strrchr) redirected to 0x402c1b0 (__GI_strrchr)
==6429== Invalid write of size 4
==6429==    at 0x80483C4: main (valgrind_illegal_read_write.c:8)
==6429==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==6429==
==6429== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

17.1.2. Variables sin inicializar

Otro tipo de problema en el que Valgrind nos puede ayudar es el de la detección de variables sin inicializar. Para ilustrar un ejemplo, el siguiente código contiene un ejemplo de variable `number` si inicializar.

```
1  // Unitialized
2  // gcc -gstabs valgrind_illegal_write.c -o valgrind_illegal_write
3  // $ valgrind  -v ./valgrind_illegal_write
4  #include <stdlib.h>
5  #include <stdio.h>
6  int main()
7  {
8    int number; //Should be initialized (i.e. to zero).
9    if (number==0)
10   {
11     printf("number is zero");
12   }
13
14   return 0;
15 }
```

A continuación se muestra lo que dice Memcheck cuando se utiliza se aplica sobre este código:

```
$ gcc -gstabs -Wall valgrind_unitialized.c -o valgrind_unitialized
$ valgrind -v --track-origins=yes ./valgrind_unitialized
```

```
==6684==
==6684== 1 errors in context 1 of 1:
==6684== Conditional jump or move depends on uninitialised value(s)
==6684==    at 0x80483F2: main (valgrind_unitialized.c:9)
==6684==  Uninitialised value was created by a stack allocation
==6684==    at 0x80483EA: main (valgrind_unitialized.c:7)
==6684==
==6684== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

17.1.3. Liberaciones ilegales de memoria

Otras de las cuestiones que comprueba Memcheck son los malos usos del comando `free`.

```
1  // illegal free
2  // gcc -gstabs valgrind_illegal_free.c -o valgrind_illegal_free
3  // $ valgrind  -v ./valgrind_illegal_free
4  #include <stdlib.h>
5  #include <stdio.h>
6  int main()
7  {
8    int * ptr=malloc(4);
9    free(ptr);
10   ptr++; //To mismatch
11   free(ptr);
12   return 0;
13 }
```

En este caso el `free` se hace sobre un puntero que no ha sido creado con `malloc`, siendo la principal causa del problema.

```
$ gcc -gstabs -Wall valgrind_illegal_free.c -o valgrind_illegal_free
$ valgrind -v --track-origins=yes ./valgrind_illegal_free
==12950== Memcheck, a memory error detector
==12950== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==12950== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==12950== Command: ./valgrind_illegal_free
==12950==
==12950== Invalid free() / delete / delete[] / realloc()
==12950==    at 0x402B06C: free (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==12950==    by 0x8048449: main (valgrind_illegal_free.c:11)
==12950==  Address 0x41fd02c is 0 bytes after a block of size 4 free'd
==12950==    at 0x402B06C: free (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==12950==    by 0x8048438: main (valgrind_illegal_free.c:9)
==12950==
==12950==
==12950== HEAP SUMMARY:
==12950==     in use at exit: 0 bytes in 0 blocks
==12950==   total heap usage: 1 allocs, 2 frees, 4 bytes allocated
==12950==
==12950== All heap blocks were freed -- no leaks are possible
==12950==
==12950== For counts of detected and suppressed errors, rerun with: -v
==12950== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

17.1.4. Fuga de memoria

Por último estaría la funcionalidad de la fuga de memoria, una de la funcionalidades más interesantes de Memcheck. La siguente pieza de código contiene un ejemplo de fuga de memoria realizada a través de una función llamada `leak`

```
1  // memory_leak
2  // gcc -gstabs valgrind_memory_leak.c -o valgrind_memory_leak
```

```
3  // $ valgrind  -v ./valgrind_memory_leak
4  #include <stdlib.h>
5  #include <stdio.h>
6  #include <string.h>
7
8  void leak()
9  {
10   void *ptr=malloc(100);
11   ptr++;
12 }
13 int main()
14 {
15   printf("#Let's leak 100 bytes");
16   leak();
17   printf("#100 bytes leaked");
18   return 0;
19 }
```

Esta es la traza de Valgrind,correspondiente al ejemplo anterior, que muestra cómo Memcheck advierte del problema:

```
$ gcc -g -Wall valgrind_memory_leak.c -o valgrind_memory_leak
$ valgrind -v --leak-check=full --show-reachable=yes  ./valgrind_memory_leak

--7436-- REDIR: 0x40cbe70 (malloc) redirected to 0x402be00 (malloc)
Let's leak 100 bytes100 bytes leaked==7436==
==7436== HEAP SUMMARY:
==7436==     in use at exit: 100 bytes in 1 blocks
==7436==   total heap usage: 1 allocs, 0 frees, 100 bytes allocated
==7436==
==7436== Searching for pointers to 1 not-freed blocks
==7436== Checked 55,964 bytes
==7436==
==7436== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==7436==    at 0x402BE68: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==7436==    by 0x8048425: leak (valgrind_memory_leak.c:10)
==7436==    by 0x8048445: main (valgrind_memory_leak.c:16)
==7436==
==7436== LEAK SUMMARY:
==7436==    definitely lost: 100 bytes in 1 blocks
==7436==    indirectly lost: 0 bytes in 0 blocks
==7436==      possibly lost: 0 bytes in 0 blocks
==7436==    still reachable: 0 bytes in 0 blocks
==7436==         suppressed: 0 bytes in 0 blocks
==7436==
==7436== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
==7436== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```



16.9.5. Emergencia: “Necesitamos la clave para entrar”

17.2. Herramienta Helgrind