ATAD | Valgrind

Valgrind is a memory mismanagement detector. It shows you memory leaks, deallocation errors, etc. Actually, Valgrind is a wrapper around a collection of tools that do many other things (e.g., cache profiling); however, here we focus on the default tool, *memcheck*.

Memcheck can detect:

- Use of uninitialised memory
- Reading/writing memory after it has been free'd
- Reading/writing off the end of malloc'd blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks – where pointers to malloc'd blocks are lost forever
- Mismatched use of malloc/calloc/realloc vs free
- Overlapping src and dst pointers in memcpy() and related functions

---

Consider the following program in `test.c`:

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  char *p;

  // Allocation #1 of 19 bytes
  p = (char *) malloc(19);

  // Allocation #2 of 12 bytes
  p = (char *) malloc(12);
  free(p);

  // Allocation #3 of 16 bytes
  p = (char *) malloc(16);

  return 0;
}
```

```
$> gcc -o test -g test.c
```

This creates an executable named `test`. To check for memory leaks during the execution of test, try

```
$> valgrind --leak-check=full ./test
```

This outputs a report to the terminal like:

```
==9704== Memcheck, a memory error detector for x86-linux.
==9704== Copyright (C) 2002-2004, and GNU GPL'd, by Julian Seward et al.
==9704== Using valgrind-2.2.0, a program supervision framework for
 ↪  x86-linux.
==9704== Copyright (C) 2000-2004, and GNU GPL'd, by Julian Seward et al.
==9704== For more details, rerun with: -v
==9704==
==9704==
==9704== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 11 from 1)
==9704== malloc/free: in use at exit: 35 bytes in 2 blocks.
==9704== malloc/free: 3 allocs, 1 frees, 47 bytes allocated.
==9704== For counts of detected errors, rerun with: -v
==9704== searching for pointers to 2 not-freed blocks.
==9704== checked 1420940 bytes.
==9704==
==9704== 16 bytes in 1 blocks are definitely lost in loss record 1 of 2
==9704==    at 0x1B903D38: malloc (vg_replace_malloc.c:131)
==9704==    by 0x80483BF: main (test.c:15)
==9704==
==9704==
==9704== 19 bytes in 1 blocks are definitely lost in loss record 2 of 2
==9704==    at 0x1B903D38: malloc (vg_replace_malloc.c:131)
==9704==    by 0x8048391: main (test.c:8)
==9704==
==9704== LEAK SUMMARY:
==9704==    definitely lost: 35 bytes in 2 blocks.
==9704==    possibly lost:   0 bytes in 0 blocks.
==9704==    still reachable: 0 bytes in 0 blocks.
==9704==         suppressed: 0 bytes in 0 blocks.
```

Let's look at the code to see what happened.

- Allocation #1 (19 byte leak) is lost because p is pointed elsewhere before the memory from Allocation #1 is free'd. To help us track it down, Valgrind gives us a stack trace showing where the bytes were allocated. In the 19 byte leak entry, the bytes were allocate in `test.c`, line 8.
- Allocation #2 (12 byte leak) doesn't show up in the list because it is free'd.
- Allocation #3 shows up in the list even though there is still a reference to it (p) at program termination. This is still a memory leak! Again, Valgrind tells us where to look for the allocation (`test.c` line 15).

Valgrind can detect many kinds of errors. Here's an explanation of the various error messages.

Explanation of error messages from Memcheck

Despite considerable sophistication under the hood, Memcheck can only really detect two kinds of errors, use of illegal addresses, and use of undefined values. Nevertheless, this is enough to help you discover all sorts of memory-management nasties in your code. This section presents a quick summary of what error messages mean.

Illegal read / Illegal write errors

For example:

```
Invalid read of size 4
   at 0x40F6BBCC: (within /usr/lib/libpng.so.2.1.0.9)
   by 0x40F6B804: (within /usr/lib/libpng.so.2.1.0.9)
   by 0x40B07FF4: read_png_image__FP8QImageIO (kernel/qpngio.cpp:326)
   by 0x40AC751B: QImageIO::read() (kernel/qimage.cpp:3621)
   Address 0xBFFFF0E0 is not stack'd, malloc'd or free'd
```

This happens when your program reads or writes memory at a place which Memcheck reckons it shouldn't. In this example, the program did a 4-byte read at address `0xBFFFF0E0`,

somewhere within the system-supplied library `libpng.so.2.1.0.9`, which was called from somewhere else in the same library, called from line 326 of qpngio.cpp, and so on.

Memcheck tries to establish what the illegal address might relate to, since that's often useful. So, if it points into a block of memory which has already been freed, you'll be informed of this, and also where the block was free'd at. Likewise, if it should turn out to be just off the end of a malloc'd block, a common result of off-by-one-errors in array subscripting, you'll be informed of this fact, and also where the block was malloc'd.

In this example, Memcheck can't identify the address. Actually the address is on the stack, but, for some reason, this is not a valid stack address – it is below the stack pointer, %esp, and that isn't allowed. In this particular case it's probably caused by gcc generating invalid code, a known bug in various flavours of gcc.

Note that Memcheck only tells you that your program is about to access memory at an illegal address. It can't stop the access from happening. So, if your program makes an access which normally would result in a segmentation fault, you program will still suffer the same fate – but you will get a message from Memcheck immediately prior to this. In this particular example, reading junk on the stack is non-fatal, and the program stays alive.

Use of uninitialised values

For example:

```
Conditional jump or move depends on uninitialised value(s)
   at 0x402DFA94: _IO_vfprintf (_itoa.h:49)
   by 0x402E8476: _IO_printf (printf.c:36)
   by 0x8048472: main (tests/manuel1.c:8)
   by 0x402A6E5E: __libc_start_main (libc-start.c:129)
```

An uninitialised-value use error is reported when your program uses a value which hasn't been initialised – in other words, is undefined. Here, the undefined value is used somewhere inside the `printf()` machinery of the C library.

This error was reported when running the following small program:

```
int main()
{
  int x;
  printf ("x = %d\n", x);
}
```

It is important to understand that your program can copy around junk (uninitialised) data to its heart's content. Memcheck observes this and keeps track of the data, but does not complain. A complaint is issued only when your program attempts to make use of uninitialised data. In this example, x is uninitialised. Memcheck observes the value being passed to `_IO_printf` and thence to `_IO_vfprintf`, but makes no comment. However, `_IO_vfprintf` has to examine the value of `x` so it can turn it into the corresponding ASCII string, and it is at this point that Memcheck complains.

Sources of uninitialised data tend to be:

Local variables in procedures which have not been initialised, as in the example above. The contents of malloc'd blocks, before you write something there.

Illegal frees

For example:

```
Invalid free()
   at 0x4004FFDF: free (vg_clientmalloc.c:577)
   by 0x80484C7: main (tests/doublefree.c:10)
   by 0x402A6E5E: __libc_start_main (libc-start.c:129)
   by 0x80483B1: (within tests/doublefree)
   Address 0x3807F7B4 is 0 bytes inside a block of size 177 free'd
   at 0x4004FFDF: free (vg_clientmalloc.c:577)
   by 0x80484C7: main (tests/doublefree.c:10)
   by 0x402A6E5E: __libc_start_main (libc-start.c:129)
   by 0x80483B1: (within tests/doublefree)
```

Memcheck keeps track of the blocks allocated by your program with `malloc/calloc/realloc`, so it can know exactly whether or not the argument to free/delete is legitimate or not. Here, this test program has freed the same block twice. As with the illegal read/write errors, Memcheck attempts to make sense of the address free'd. If, as here, the address is one which

has previously been freed, you wil be told that – making duplicate frees of the same block easy to spot.

Author and support

Bruno Silva (bruno.silva@estsetubal.ips.pt)

You should ask your PL teacher for any help regarding these contents and procedures.