Explore Memory and Resource Leak Detection Tools

By Jeff Tranter (/author/jeff-tranter) | Wednesday, November 29, 2017

(/#twitter) (/#linkedin) (/#facebook) (/#reddit)

Memory and resource leaks are the stuff of nightmares for programmers. If a program doesn't properly free memory or other resources, it may appear to run correctly, but randomly crash or misbehave after working normally for hours or days. All too often, the problem is only discovered just before the application is supposed to be shipped to customers.

While Qt helps somewhat with it's object model, resource leaks can occur with any programming language like C++ where the user is responsible for managing memory or other resources like file handles. Let's take a look at some of the tools that can help us track down these insidious problems.

GNU malloc

Under Linux using GNU libc, the kernel and/or C run-time will sometimes detect memory allocation or usage errors without doing anything special in your code or using any external tools. It seems to have improved over what was typically detected some time in the past. Run-time errors might look like these:

```
*** Error in './a.out': free(): invalid pointer: 0x000056244b636064 ***
```

```
*** Error in './a.out': free(): invalid next size (fast): 0x000056416893e030 ***
```

This will typically be followed by a backtrace, a display of the process memory map, and a core dump (if enabled).

When using GNU libc, more checking of memory allocation using malloc and free can be enabled by setting the variable MALLOC_CHECK_. When MALLOC_CHECK_ is set, a special but less efficient implementation is used which is designed to be tolerant against simple errors, such as double calls of **free** with the same argument, or overruns of a single byte. If MALLOC_CHECK_ is set to 0, any detected heap corruption is silently ignored. If set to 1, a diagnostic is printed on standard error. If set to 2, **abort** is called immediately.

Mtrace is a builtin part of glibc which allows detection of memory leaks caused by unbalanced **malloc/free** calls. To use it, the program is modified to call **mtrace()** and **muntrace()** to start and stop tracing of allocations. A log file is created, which can then be scanned by the **mtrace** script.

There are also some functions you can call into GNU libc to check the consistency of the heap at run time. See https://www.gnu.org/software/libc/manual/html_node/Heap-Consistency-Checking.html (https://www.gnu.org/software/libc/manual/html_node/Heap-Consistency-Checking.html) for more details.

Valgrind memcheck

The above checks help with errors in memory access and usage, but errors related to unfreed resources (resource leaks) will go undetected. If you suspect a resoorce leak, it is time to get out the big guns. The codemiere tool for doing this is valgrind's **memcheck** tool.

Memcheck is the default tool when running valgrind, and will detect a nmber of problems that can occur in C and C++ programs:

- Accessing memory you shouldn't, e.g. overrunning and underrunning heap blocks, overrunning the top of the stack, and accessing memory after it has been freed.
- Using undefined values, i.e. values that have not been initialised, or that have been derived from other undefined values.
- Incorrect freeing of heap memory, such as double-freeing heap blocks, or mismatched use of malloc/new/new[] versus free/delete/delete[]
- Overlapping source and destination pointers in memcpy and related functions.
- Passing a negative value to the size parameter of a memory allocation function.
- Memory leaks.

Here is the output from a very simple C++ program that has a couple of memory leaks where allocated memory is not being freed:

```
% valgrind --leak-check=full ./a.out<br />
==8110== Memcheck, a memory error detector<br />
==8110== Copyright (C) 2002-2015, and GNU GPL d, by Julian Seward et al.<br />
==8110== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info<br />
==8110== Command: ./a.out<br />
==8110== Hello, world!<br />
==8110==<br />
==8110== HEAP SUMMARY:<br />
==8110== in use at exit: 56 bytes in 2 blocks<br />
==8110== total heap usage: 4 allocs, 2 frees, 73,784 bytes allocated<br />
==8110==<br />
==8110== 16 bytes in 1 blocks are definitely lost in loss record 1 of 2<br />
```

```
==8110== at 0x4C2DB2F: malloc (in /usr/lib/valgrind/vgcodeload_memcheck-amd64-linux.so)
==8110== by 0x108A04: main (demo.cpp:18)<br />
==8110==<br />
==8110== 40 bytes in 1 blocks are definitely lost in loss record 2 of 2<br />
==8110== at 0x4C2E8BF: operator new[](unsigned long) (in /usr/lib/valgrind
/vgcodeload_memcheck-amd64-linux.so) ==8110== by 0x1089C9: main (demo.cpp:7) ==8110==
==8110== LEAK SUMMARY:<br />
==8110== definitely lost: 56 bytes in 2 blocks ==8110== indirectly lost: 0 bytes in 0
blocks ==8110== possibly lost: 0 bytes in 0 blocks<br />
==8110== still reachable: 0 bytes in 0 blocks<br />
==8110== supcodessed: 0 bytes in 0 blocks<br />
==8110== ==8110== For counts of detected and supcodessed errors, rerun with: -v<br />
==8110== ERROR SUMMARY: 2 errors from 2 contexts (supcodessed: 0 from 0)
```

Finding memory leaks with valgrind is often not trivial, and in large programs you typically need to ignore a number of false positives in third party libraries.
Adequately covering valgrind could easily take a whole series of blog posts on its own. You can read the documentation or start any of the many tutorials to be found on the Internet.

Dmalloc

Dmalloc or *Debug Malloc Library* is a drop in replacement for the system's **malloc** and other memory management functions. It includes facilities for debugging errors and memory leaks. It runs on most operating system platforms.

I haven't personally used it, and it appears not to have been updated sine 2007, so I can't confirm how well it works on current systems. For more details see http://dmalloc.com/ (http://dmalloc.com/).

Electric Fence

Electric Fence is another drop in replacement memory allocation library. It uses the virtual memory hardware of your computer to place an inaccessible memory page immediately after or before each memory allocation. This will generally cause your program to immediately seg fault if it attempts to read or write outside of the range of allocated memory. To use it, link your program with the **efence** library (e.g. with the **-lefence** linker option) or set the LD_codeLOAD environment variable to load the library when running your program. Here is the output for a small example program I ran:

```
% LD_codeLOAD=libefence.so.0.0 ./a.out<br />
Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens <<a
href="mailto:bruce@perens.com">bruce@perens.com</a>><br />
Hello, world!<br />
Segmentation fault (core dumped)
```

Looking at the core dump in the debugger indicated where the problem occured, as shown in this partial debug session below:

```
% gdb a.out core GNU gdb (Ubuntu 7.12.50.20170314-0ubuntu1.1) 7.12.50.20170314-git<br />
Core was generated by './a.out'.<br />
Program terminated with signal SIGSEGV, Segmentation fault.<br />
#0 0x00005598e889ea1f in main () at demo.cpp:11<br />
11 x[i] = 1; (gdb)
```

Electric Fence should be available on most Linux distributions. On Ubuntu Linux it is provided in the package electric-fence. For more details see http://elinux.org/Electric_Fence (http://elinux.org/Electric_Fence).

Dbgmem

Dbgmem is a memory debugger for C and C++ programs on Linux systems. It can help track down memory leaks, heap memory corruption, stack corruption and use of freed or uninitialized heap memory. It works by overriding the Glibc memory allocation, memory and string manipulation functions. I haven't personally used it, but it looks interesting. More details can be found at http://dbgmem.sourceforge.net/ (http://dbgmem.sourceforge.net/)

Memwatch

This is another memory leak and allocation tool. I haven't tried it, and it hasn't been touched since 2003, so I'm skeptical that it works on current systems but you may want to give it a try. See http://www.linkdata.se/sourcecode/memwatch/ (http://www.linkdata.se/sourcecode/memwatch/)

Mpatrol

This is another memory allocation testing and debugging library. It also looks a little old, not having been updated since 2008. See http://mpatrol.sourceforge.net/ (http://mpatrol.sourceforge.net/) for details.

Sar

The System Activity Reporter, or **sar** command, displays the contents of selected cumulative activity counters in the operating system. Information is reported on the paging system, i/o, CPUs, network interfaces, swap space, file system, and others. Here is a portion of the output when run with the -A option to display all output, and specifying an interval of one second and a count of one:

```
% sar -A 1 1<br />
Linux 4.10.0-35-generic (xpslaptop) 2017-09-19 _x86_64_ (8 CPU)<br />
04:37:21 PM CPU %usr %nice %sys %iowait %steal %irq %soft %guest %gnice %idle<br />
04:37:22 PM all 11.67 0.00 2.13 0.00 0.00 0.00 0.13 0.00 0.00 86.07<br />
04:37:21 PM proc/s cswch/s 04:37:22 PM 4.00 7568.00<br />
04:37:21 PM INTR intr/s 04:37:22 PM sum 1265.00<br />
04:37:21 PM CPU MHz<br />
04:37:22 PM all 899.93 04:37:22 PM 0 899.93<br />
04:37:22 PM 1 899.93 04:37:22 PM 2 899.93<br />
04:37:22 PM 3 899.93 04:37:21 PM TEMP degC %temp DEVICE<br />
04:37:22 PM 1 36.00 36.00 coretemp-isa-0000<br />
04:37:22 PM 2 34.00 34.00 coretemp-isa-0000<br />
04:37:22 PM 3 34.00 34.00 coretemp-isa-0000 Average: kbmemfree kbmemused %memused kbbuffers
kbcached kbcommit %commit kbactive kbinact kbdirty kbanonpg kbslab kbkstack kbpgtbl
kbvmused<br />
Average: 2926136 13341536 82.01 1448812 7238924 10483736 29.96 6586924 5648208 1444 3463220
723676 15728 85884 0
```

Miscellaneous

As well as the specific tools I've mentioned, there are a number of standard Linux tools for looking at memory and resource usage that you should be familiar with. I've listed here the ones that I've found are commonly used. They are typically installed out of the box in standard Linux distributions, or available as standard packages. You can read the relevant documentation (typically starting with the man pages) to learn how to use them.

df - report file system disk space usage
dstat - versatile tool for generating system resource statistics
du - estimate file space usage
find - search for files in a directory hierarchy
free - display amount of free and used memory in the system
ifstat - report interface statistics
iostat - report CPU statistics and input/output statistics for devices and partitions
lsof - list open files
memstat - identify what's using up virtual memory
mpstat - report processors related statistics
pmap - report memory map of a process
ps - report a snapshot of the current processes
top - display Linux processes
vmstat - report virtual memory statistics