Precise detection of memory leaks

Jonas Maebe Michiel Ronsse Koen De Bosschere

Ghent University, ELIS Department
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
jmaebe|ronsse|kdb@elis.UGent.be
http://www.elis.UGent.be/diota

Abstract

A memory leak occurs when a program allocates a block of memory, but does not release it after its last use. In case such a block is still referenced by one or more reachable pointers at the end of the execution, fixing the leak is often quite simple as long as it is known where the block was allocated. If, however, all references to the block are overwritten or lost during the program's execution, only knowing the allocation site is not enough in most cases. This paper describes an approach based on dynamic instrumentation and garbage collection techniques, which enables us to also inform the user about where the last reference to a lost memory block was created and where it was lost, without the need for recompilation or relinking.

1 Introduction

A memory leak is a memory management problem which indicates a failure to release a previously allocated memory block. The term can be used in two contexts. The first is when indicating imperfections in garbage collectors as used in e.g. Java Virtual Machines, in case they missed the fact that a previously allocated block is not referenced anymore and thus is not added to the pool of free blocks.

The second context is when the programmer himself is responsible for explicitly freeing all blocks of memory that he allocated. This is still the case in most run time environments today and also the situation which we will focus on in this paper.

Leaking blocks of memory during a program execution has several negative consequences. It often results in said program acquiring more and more memory from the operating system during its execution.

As such, overall system performance will degrade over time, as allocated but unused blocks of memory will have to be swapped out once the system runs out of free physical memory. Eventually, a program may even exhaust its available virtual address space, which will cause it to terminate due to an out-of-memory error.

Several packages that can perform memory leak detection already exist. The necessary instrumentation can happen at different levels. Insure++ [5] rewrites the source code of an application. Many leak detectors operate at the library level by intercepting calls to memory management routines, such as in case of LeakTracer [1], memdebug, memprof and the Boehm Garbage Collector [2].

Finally, it is possible to instrument at the machine code level. Purify [8] statically instruments the object code of an application and the libraries it uses. Dynamic instrumentors such as Valgrind [7] delay the instrumentation until run time.

Except for Insure++, all of the mentioned debugging helpers only tell the programmer where the leaked block of memory was allocated, but not where it was lost. Insure++ does show where the last pointer to a block of memory was lost, but not where this pointer got its value. Additionally, since it is a source code instrumentation tool, it requires recompilation and cannot provide detailed information about leaks in third-party libraries of which the source code is unavailable.

In this paper, we present a technique that uses dynamic instrumentation at the machine code level to track all pointers to allocated blocks of memory. It is completely language- and compiler-independent and can show where the leaked blocks were allocated, lost and where the last references to these blocks were created.

In what follows, we first give a short overview of the instrumentation framework we use. Next, we discuss the kinds of memory leaks that exist and how they may occur. We then describe in great detail how we can detect these leaks, as well as some implementation details. Finally, we conclude after presenting a short evaluation and discussing our future plans.