Subject: Re: Suppressions file for valgrind
Posted by Novo on Wed, 04 Jun 2008 16:59:11 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Wed, 04 June 2008 06:15
Well... you're right, I had to add some suppressions for different libraries on ubuntu 8.04. But, well, suppressing non-existent libraries does no harm

number of suppressions and length of stack trace in each suppression.

BTW, I've found an interesting message in valgrind mailing list:

One of the kinds of errors that Memcheck finds is dangerous uses
of uninitialised values, usually when they get used in an
if-statement. This is a useful facility, but it is also the
single most complained about aspect of Memcheck. The problem is
that the point where Memcheck reports the error is often a long
way after the heap or stack allocation that created the
uninitialised value. And so it can be very difficult to find the
root cause of the problem.

Memcheck will now optionally track these origins, and, when
reporting an uninitialised value error, it can show the origin
too. If the origin is a heap allocation, it shows where the
block was allocated. If the origin was a stack allocation, it
will tell you the function that did the allocation. A couple of
simple examples are shown below.

Of course there's no free lunch: Memcheck's speed is
approximately halved, and memory consumption increases by a
minimum of 100MB. But in initial testing, on a large C++
codebase, it has proven effective.

You can try out this functionality using the SVN trunk:

```
  svn co svn://svn.valgrind.org/valgrind/trunk
  cd trunk
  ./autogen.sh
  ./configure --prefix=...
```

and then run with --track-origins=yes.

This functionality was inspired by the work of Bond, Nethercote,
et al, as reported in the paper "Tracking Bad Apples: Reporting
the Origin of Null and Undefined Value Errors"

(http://www.valgrind.org/docs/origin-tracking2007.pdf), but the
actual implementation is very different from that described in
the paper.

Feedback, comments, bug reports welcome.

J

Simple example of an uninitialised value use originating from
a heap block:

```
  Conditional jump or move depends on uninitialised value(s)
     at 0x400ACB: main (origin1-yes.c:64)
   Uninitialised value was created by a heap allocation
     at 0x4C234BB: malloc (vg_replace_malloc.c:207)
     by 0x400A9B: main (origin1-yes.c:61)
```

And one from a stack allocation (a local, or "auto" variable):

```
  Conditional jump or move depends on uninitialised value(s)
     at 0x400944: main (origin3-no.c:33)
   Uninitialised value was created by a stack allocation
     at 0x4008B4: main (origin3-no.c:15)
```

An example from the opposite end of the size/triviality spectrum:

```
  Use of uninitialised value of size 8
     at 0x4F277E7: BitmapReadAccess::SetPixelFor_24BIT_TC_BGR
                   (bmpacc2.cxx:195)
     by 0x4F1A8CE: Bitmap::ImplConvertUp (bmpacc.hxx:542)
     by 0x4F1B9D3: Bitmap::Convert (bitmap3.cxx:365)
   Uninitialised value was created by a heap allocation
     at 0x4C22DDB: malloc (vg_replace_malloc.c:207)
     by 0x719B4FA: rtl_allocateMemory (alloc_global.c:311)
     by 0x470BDA: allocate (operators_new_delete.cxx:160)
     by 0x470C4A: operator new[](unsigned long)
                  (operators_new_delete.cxx:239)
     by 0xEBCAABC: X11SalBitmap::ImplCreateDIB (salbmp.cxx:194)
```