When the Stars Align, and When They Don't

TIM HOCKIN

Tim Hockin attended Illinois State University to study fine art, but emerged with a BS in computer science instead. A fan of Linux since 1994, he joined Cobalt Networks after graduating and has surfed the Linux wave ever since. He joined Google's nascent Platforms group in 2004, where he created and lead the system bring-up team. Over time he has moved up the stack, most recently working as a tech lead (and sometimes manager) in Google's Cluster Management team, focusing mostly on node-side software. He is somewhat scared to say that his code runs on every single machine in Google's fleet. thockin@google.com

I got my first real bug assignment in 1999, at the beginning of my first job out of undergrad. Over a period of weeks, I went through iterations of code-reading, instrumentation, compiling, and testing, with the bug disappearing and reappearing. In the end, the problem turned out to be a subtle misalignment, something I would never have guessed when I started out.

Here's how I got started: A user reported that their database was being corrupted, but only on our platform. That's entirely possible—our platform was, for reasons that pre-dated my employment, somewhat odd. It was Linux—easy enough—but on MIPS. The MIPS port of Linux was fairly new at the time and did not have a huge number of users. To make life more fun, we used MIPS in little-endian mode, which almost nobody else did. It was supported by the toolchain and the kernel, so we all assumed it was workable. To find a bug on this rather unique platform was not surprising to me.

I took to the bug with gusto. Step one, confirm. I reproduced the user's test setup—a database and a simple HTML GUI. Sure enough, when I wrote a number, something like 3.1415, to the database through the GUI and then read it back through the GUI, the software produced a seemingly random number. Reloading the page gave me the same number. Problem reproduced.

With the arrogance of youth, I declared it a bug in the customer's GUI. I was so convinced of this that I wanted to point out exactly where the bug was and rub their noses in it. So, I spent a few hours instrumenting their HTML and Perl CGI code to print the values to a log file at each step of the process as it was committed to the database. To my great surprise, the value was correct all the way up until it was written to the database!

Clearly, then, it was a bug in the database. I downloaded source code for the database (hooray for open source!). I spent the better part of a day learning the code and finding the commit path. I rebuilt the database with my own instrumentation, and fired it up. Lo and behold, the bug was gone! The value I wrote came back perfectly. I verified that the code I was testing was the same version as in our product—it was. I verified that we did not apply any patches to it—we didn't.

Clearly, then, it was a bug in the toolchain. I spent the next week recompiling the database (to the tune of tens of minutes per compile-test cycle) with different compiler versions and flags, with no luck—everything I built worked fine. Someone suggested that maybe the Linux packaging tool (RPM) was invoking different compiler behavior than I was using manually. Great idea, I thought. I hacked up a version of the package that used my instrumented code and ran that. Bingo—the bug was back.

I began dissecting all of the differences I could figure out from the RPM build and my manual build. After a number of iterations over a number of days, I tracked it down to a single -D flag (a #define in the C code) to enable threading in the database. I was not setting this when I compiled manually, but the RPM build was.

Clearly, then, the bug was in the database's threading code. Suddenly it got a lot more daunting but also more fun. I spent even more time auditing and instrumenting the database code for threading bugs. I found no smoking gun, but iterative testing kept pushing the problem further and further down the stack. Eventually, it seemed that the database was innocent. It was writing through to the C library correctly and getting the data back corrupted.

Clearly, then, the bug was in the C library. Given the circumstances of our platform, this scenario was believable. I downloaded the source code for GNU libc and started instrumenting it. Compiling glibc was a matter of hours, not minutes, so I tried to be as surgical as I could about changes. But, every now and then, something went awry in the build and I had to "make clean" and start over—it was tedious. Several iterations later, libc was absolved of any wrongdoing—the data got all the way to the write() syscall correctly, but it was still incorrect when I read and printed it later.

Clearly, then, the bug was in the kernel. If I thought iterating the C library was painful, iterating the kernel was agony. Hack, compile, reboot, test, repeat. I instrumented all along the write() path all the way down to the disk buffers. Now several weeks into this bug, the kernel was free from blame. Maybe the data wasn't actually getting corrupted? Why this idea did not hit me before I blame on my initial overconfidence, which was much diminished by this point.

I took a new tack—could I reproduce it outside of the database test? I wrote a small program that wrote a double precision floating point value (the same as the test case) to a file, read it back, and printed it out. I linked the threading library, just like the database code, but the bug did not reproduce. That is, until I actually ran the test case in a different thread. Once I did that, the result I printed was similarly, but not identically, corrupted. I tried with non-float values, and the bug did not reproduce. I dredged my memory for details of the IEEE754 format and confirmed that the file on disk was correct—the corruption was happening in the read path, not the write path!

Given this fresh information, I instrumented the test program to print the raw bytes it had read from the file. The bytes matched the file. But when I printed the number, it was wrong. I was getting close!

Clearly, then, the bug was in printf(). I turned my attention back to the C library. I quickly found that the bytes of my float value were correct before I called printf() and were incorrect inside printf(). The printf() family of functions uses C variable-length argument lists (aka varargs)—this is implemented with compiler support to push and pop variables on the call stack. Because this process is effectively manual, printf() must trust you, the programmer, to tell it what type you pushed. It dawned on me that if I told printf() that I pushed a float when I really pushed an int, it would interpret the data that it popped incorrectly—leading to exactly this type of corruption. I verified that manually doing varargs of a double precision value corrupted the value.

To recap: the problem only occurs with (1) floating point values, (2) running in a thread, (3) with varargs.

Clearly, then, the bug was in the varargs implementation. I puzzled through the varargs code—a tangle of macros to align the memory access properly (16 bytes required for double), and I could find no flaw with the code. It followed the platform's spec, but the resulting value was still wrong. I looked at the disassembled code for pushing and popping the values, and it looked correct—the alignment was fine. In desperation, I instrumented the calling code to print the stack address as it pushed the argument and compared that with the address that varargs popped—they did not match. They were off by eight bytes!

After many hours of staring at disassembled code and cross-referencing the spec, it became clear—the logic used when pushing the value onto the stack was subtly different than the logic used to pop it off. Specifically, the push logic was using offsets relative to the frame pointer, but the pop logic used the absolute address. In the course of reading the ABI specification, I noticed that almost every value on the stack is required to be double-word (8 byte) aligned, the stack frame itself is required to be quad-word (16 byte) aligned. When I printed the frame pointer register, I found that it failed to meet this specification.

It seemed obvious: All I had to do was find the code that allocated the stack space and align it better.

I proceeded to take apart all of the LinuxThreads code related to stack setup. It allocated a block of memory for the thread stack, but the allocation was already aligned. Ah, but stacks grow down! The code placed an instance of a thread descriptor structure at the top of the stack region (base+size—sizeof(struct)) and initialized the stack to start below that. On a lark, I printed the size of that structure—it was a multiple of eight in size, but not a multiple of 16.

Gotcha!

I added a compiler directive to align the structure to 16 bytes, which has the side effect of making the size a multiple of 16, and compiled the C library one more time. The problem was gone, and the database's GUI now showed correct results. All of this work, and the fix was to add eight bytes of empty space to a structure. With a sigh, I dashed off a patch to the MIPS maintainers for GNU libc, which was met with a response to the effect of, "Oh! We've been hunting that one years." It felt like I had been, too.