Reliable Code

The Value of Doubt

Gerard J. Holzmann

For me, software engineering is interesting because it so convincingly gives the illusion of being precise. When you write software, there's technically no room for ambiguity. Once you manage to get your code to compile, you can be sure the machine will relentlessly execute the true meaning of that code. What makes it especially interesting is that we often find we have a difference of opinion with the machine about what that true meaning is. We always lose those battles, of course. We can specify things precisely enough that a dumb machine can execute them, but when things don't work the way we meant, we can never blame the machine. In almost all cases the mistake will be our own.

Does all this make software developers more aware of their own fallibility than other people? Programmers learn to doubt everything. You can be convinced that the code you wrote is absolutely correct, until you wake up in the middle of the night three weeks later and realize you missed something really important. Doubt is probably key to becoming a good programmer. If you don't doubt the correctness of your work, you have no incentive to look for the hidden spoilers that always seem to be there. As the physicist Richard Feynman once said, "[Only] if you know that you are not sure, you have a chance to improve the situation."[1]

The Friday Fix

You might think that once you've found a bug, either by waking up in the middle of the night or by careful analysis and testing, your troubles are over. You can fix the bug and move on with your life, right? Alas, that's not the case. Every bug fix runs a risk of introducing a new bug that might be just as bad as the one you tried to fix. The chance that this happens increases with the amount of time that has passed since the code was first written, but it seems to average to about one new bug introduced for every seven bugs fixed. Some studies have found that the risk of introducing new bugs even depends on the day of the week, with Friday being the worst.[2] It's not just programmers who tend to become a little more reckless as the weekend draws near. A study a few years back found that your chances of dying from surgery complications are also higher if you enter the hospital on a Friday.[3]

In a well-run software project, each newly discovered bug will trigger a root-cause analysis that identifies not only where the bug originated but also why the defensive mechanisms that were in place (such as code reviews, simulations, tests, and various forms of analysis) failed to catch it. With our doubting-Thomas cap on, we should continue to ask, are we sure we identified the real root cause, or could other contributing factors exist? And, if a fix is attempted, are we really sure it's the right fix and it's going to be applied correctly? Sadly, a proper dose of doubt is often missing. Let's look at some examples.

The Day the Network Broke

Not many people remember January 1990 as having any special historical significance, but it held two real stingers that shook the belief that the networks connecting us are robust to failure. On 22 January, Robert Morris Jr., then a Cornell University graduate student, became the first person to be convicted under the US Computer Fraud and Abuse Act. A little over a year earlier, in November 1988, Morris had unwittingly crippled the then still-fledgling Internet with a specially crafted virus that has become known as the Morris worm. Morris had merely wanted to find a simple way to count the number of computers connected to the network at that time. He was as surprised as everyone else that his code ended up crippling it.

Morris's worm exploited some basic security flaws he had discovered in networking software. Everyone knew at the time that flaws such as buffer overruns were a potential problem, but not many people realized what the consequences of those flaws could be. That a student could accidently bring down an entire computer network was a new phenomenon.

The second event in January 1990 was perhaps even more disruptive. In this case, a minor coding flaw in a revised fragment of AT&T switching code took down the entire US long-distance telephone network for the better part of a day. The revised software had been running in the switches for a while, but the right circumstances that could trigger the bug simply hadn't happened yet. That moment came at 1:23 p.m. Eastern Standard Time on 15 January 1990 (Martin Luther King Day), when a switch in New Jersey experienced a sudden burst of traffic that exposed the fault.

The phone system is, of course, designed to handle service disruptions. One way it does so is by rebooting the switch that has the problem (sound familiar?). When the switch comes back online, it typically has some backup work to take care of, which means sending a little burst of extra traffic to its neighboring switches. In this case, those little bursts of traffic triggered the bug in every one of the neighboring switches, which passed the problem on to their neighbors, and so on. The resulting domino effect saw every switch in the US long-distance telephone network crash, reboot, resume work, and crash again, in a seemingly never-ending cycle. Within 20 minutes of the initial problem, about half of all US long-distance calls were failing. It was the most disastrous service disruption AT&T had ever experienced.

Tiny Flaws

A few days after the event, AT&T chairman Bob Allen sent a note to all employees (I was one of them) saying in part, "One tiny, isolated flaw in our network caused astonishingly far-reaching consequences. … Tiny, seemingly insignificant flaws in what we do—in *all* corners of the company—can have far-reaching impact on customers who rely upon us. That's why we must ensure quality in everything we do."

So, what was the flaw? Identifying the root cause of the problem took a few days. The search first examined in detail the last set of changes that had been made to the otherwise well-functioning code controlling the network switches. The culprit was a piece of code that looked like this before the change:

```
for (…) {
  switch (X) {
  case A: …; break;
  case B: …; break;
  }
  …
}
```

This code fragment was rewritten to replace the `switch` statement with a simpler `if-then-else`, because there were only two cases to distinguish anyway. The new code fragment looked like this:

```
for (…) {
  if (X == A) {
    …
  } else {
    …
    break;
  }
  …
}
```

Many of today's coding standards require that `switch` statements should indeed have more than two cases, so the change was reasonable. But you probably noticed in the second fragment that the developer forgot to remove one of the `break` statements. The compiler didn't complain; the `break` statement now simply terminated the surrounding `for` loop, skipping all the required processing that followed the `if-then-else`. That was the bug.

The mistake was discovered by the same developer who had revised that part of the code a few weeks earlier. When it was reported back up the, very substantial, management chain, a remedy had to be put in place to ensure that nothing like this could happen again. This could have involved a range of things, such

as strengthening the code review process and adopting more rigorous software test methods. The key observation was that the vetting of code revisions was not sufficiently rigorous. Put differently, there wasn't enough doubt in the system that software fixes might themselves be suspect. One recommendation made in response to the events of 15 January stood out. A not very software-savvy manager asked the development teams to justify every single `break` statement that was used in the tens of millions of lines of switching code. This, of course, was a rather laborious way of accomplishing exactly nothing.

Crying Wolf

Our collective awareness of security-related vulnerabilities has improved a lot since the early nineties. Vetting code for hidden flaws that hackers can exploit is hard, so it's understandable that sometimes people try to find shortcuts. One company I'm familiar with requires scanning all its critical code with a popular open source tool that performs a rough audit of security vulnerabilities. The main attraction of the tool could be that it's freely available. However, as best as I can tell, it restricts itself to just listing every call to a small set of library functions that might be misused in poorly written code. Curiously, the tool doesn't attempt to analyze whether those calls are indeed vulnerable to attack.

As one simple example, most developers know to avoid the following type of call to copy a string:

```
strcpy(a, b);
```

If `a` is an array of known size, a first attempt to make this safer is to use a bounded version of the routine:

```
strncpy(a, b, sizeof(a));
```

The new call restricts the number of characters that can be copied to the number that can actually fit in the destination buffer. This still isn't sufficient, though, because when the string pointed to by argument `b` is longer than the size of `a`, the substring that ends up in buffer `a` won't be null-terminated. So any attempt to read the contents will still lead to a buffer overflow.

There are several ways to protect against this. If you're sure the problem could never occur, the right thing to do might be to insert an assertion just before the call:

```
assert(strlen(b) < sizeof(a));
```

Another way is to clear the target buffer first and copy one fewer character than the maximum:

```
memset(a, 0, sizeof(a));
strncpy(a, b, sizeof(a)-1);
```

A third, more efficient, way is to set at least the last element of the destination buffer to zero:

```
strncpy(a, b, sizeof(a));
a[sizeof(a)-1] = '\0';
```

In this case, the compiler can precompute the array reference because the index is a constant expression, so that the final assignment adds just one instruction.

The security analysis tool I mentioned, in deference to the fact that `strncpy` is somewhat safer than `strcpy`, issues high-severity warnings only for the latter calls. In neither case does it seem to analyze the context. Seeing this, I deviously decided to run the tool on its own source code, to see how well the tool developers managed to avoid triggering so many false alarms. This experiment produced a rewarding list of 219 warnings, 91 of which the tool classified as high severity. Although this number was substantial, it was fewer than I expected to find in a few thousand lines of code, so I looked in a little more detail at the use of string copy routines in the code.

The source code of the tool contained three calls to the maligned `strcpy` routine, with just two triggering high-severity warnings in the output. The third call got a free pass, for unknown reasons. The

code also contained eight calls to `strncpy`. Those eight calls nicely skirted triggering high-severity warnings, but, amazingly, seven of them were written like this:

```
strncpy(a, b, strlen(b));        (1)
```

which is functionally identical to the maligned version,

```
strcpy(a, b);      (2)
```

If variable `b` points to a longer string than buffer `a` can hold, this `strncpy` call will happily write beyond the extent of array `a`. Did that earlier manager somehow move to the company that produced this tool and ask that all calls of type 2 be replaced with calls of type 1? Also in this case, the change accomplished nothing more than to suppress some tool warnings.

Point MisTaken

Here's another small example of attempts to cleverly defeat warnings but not heed them. Modern static analyzers are very good at catching potential null-pointer derefencing errors. Such errors occur when programs try to retrieve a value stored at a given address but the address pointed to is zero. Normally, address zero is in a nonaddressable part of memory reserved for the OS. Globally declared pointers in C programs are by definition initialized to zero, so the error is usually caused by a failure to initialize a pointer before it's first used.

These warnings can be of great value, but it's much easier to get rid of the warning (the symptom) than the underlying bug (the cause). To get a clean slate from the static analyzers, which most managers expect, developers sometimes initialize pointers with dummy values, as in this made-up example:

```
int x;
int *ptr = &x;
```

The static analyzer will no longer issue a warning when the pointer `ptr` is first dereferenced. But the bug is still there and will now be a little harder to diagnose when the error strikes.

The last two examples illustrate nicely that even when the warnings from compilers or static analyzers are disappearing, it might not quite be time to uncork the champagne. It might be time to check precisely how the warnings were made to disappear. As Feynman wrote, "It is important to doubt. … Doubt is not a fearful thing, but a thing of very great value."[1]

References

1. R. Feynman, "The Uncertainty of Science," *The Meaning of It All: Thoughts of a Citizen-Scientist*, Addison-Wesley, 1998, pp. 1–28.
2. J. Śliwerski, T. Zimmerman, and A. Zeller, "Don't Program on Fridays! How to Locate Fix-Inducing Changes," 2005; thomas-zimmermann.com/publications/files/sliwerski-wsr-2005.pdf.
3. P. Aylin, "Day of Week of Procedure and 30 Day Mortality for Elective Surgery: Retrospective Analysis of Hospital Episode Statistics," *British Medical J.*, 2013; www.bmj.com/content/346/bmj.f2424.

Gerard J. Holzmann *works at the Jet Propulsion Laboratory on developing stronger methods for software analysis, code review, and testing. Contact him at gholzmann@acm.org.*