A Three-Step Program for Recovering Hackers

Gerard J. Holzmann
Laboratory for Reliable Software

Following a three-step program can help software developers rely less on users to catch their bugs.

Many of us learned to write software in simpler times, and have perhaps grown a little too comfortable with a relaxed style of software development. We write the code, check that it can do what we intended, and move on. Except, somehow, we never really seem to be done with the code. Our users have a nasty habit of focusing on the weak spots and then peppering us with bug reports. If you're of a sunny disposition, you can take the number of reports as a positive indication that people are using your code, but your boss might have a less favorable interpretation. You might catch yourself giving long explanations about the thoroughness of your design and how the users are just confused about how it works. If so, then you are in the denial phase. At some point, though, the weaknesses in your code will come into sharper focus, and you might get annoyed: anger. You might toy with the idea of telling your boss that the user manuals need improvement or that you need a faster workstation: bargaining. It's difficult to come to terms with the notion that perhaps your own workmanship needs improvement. However, slowly but surely, maybe after navigating a mild depression phase, you get to where you need to be to move forward: acceptance. This column describes what you can do next.

Most software applications have bugs. Many of those that show up after testing (often referred to more cheerfully as *residual defects*) are merely annoying, but occasionally a bug can turn out to be a showstopper. But the showstopper bugs aren't the only ones that can do harm: an unusually large number of bugs that fall into the "mere annoyance" category can impact your reputation. Let's go over some simple steps that recovering hackers can use to improve their code's quality. Our recovery program starts with three simple, but habit-forming, steps: tally, target, and track.

STEP ONE: TALLY ME WHERE IT HURTS

The recovery process begins by making a general assessment of the quality of your work. This means tallying up some simple code metrics. You can start by measuring five specific numbers that will tell you something about your code.

First, measure the number of lines of code you have, but don't count blank lines or comments. You want to count every line that could in principle contain a bug (for a suitable code counter, see for instance http://spinroot.com/gerard/ncsl.tar.gz).

Second, count the number of assertions in your code: a simple "`grep -c assert *.c`" might do. You aren't using assertions? That's simple: your number on this metric will be zero.

For the next measurement you want to look at the results of a standard compilation of your code, just like you always do for production, but with one small change: use the most up-to-date version of your compiler. If the compiler you're using doesn't have a version that dates within the past 12 months, switch to one that does. The best compilers are actively being worked on, and they tend to get a lot better with every new release. Use, for example, gcc or

 Published by the IEEE Computer Society

g++ compilers: they aren't just free, they have very large numbers of users, and they're very good. Your code should, of course, be language-compliant, and it shouldn't rely on any one compiler's special quirks or peculiarities. Yes, this rule applies even if you're developing embedded software for a specific hardware device.

For the third metric, measure the number of warnings the compiler issues when it compiles your code. For this measurement, you should tell the compiler which specific language standard you intend to comply with, for instance, by adding the compiler `flag -std=c99` to gcc.

For the fourth metric, add two additional compilation flags, `-Wall` and `-pedantic`, and repeat the build. Again, count the number of warnings issued.

And finally, for the fifth and last number, use any commercially available static source code analyzers (http://spinroot.com/static/) and run it on your code. For the most part, static analyzers work just like a compiler, issuing warnings on the code. Count the number of warnings generated.

To complete the tally steps, normalize the number in each category by the line count, to get the numbers per one thousand lines of noncomment source code, or KNCSL.

We could, of course, add many other quality-related metrics here, but gathering these first few numbers will be a good start.

STEP TWO: TARGET IMPROVEMENT

Now that you've tallied some metrics, it's time to set a target for improving the code.

A first target will be to reach the point where you have zero warnings when compiling your code for the third metric. Zero compiler warnings are both a mark of good workmanship and an important benchmark to improve your code's maintainability.

If adding or modifying any part of the code generates new warnings, you want them to stand out prominently so that you can give them careful attention. If a build routinely generates hundreds or thousands of warnings, the new warnings that show up are virtually unnoticeable and almost certainly escape attention.

If your initial metrics tell you that you're far removed from this target, you might want to set a series of subgoals. Depending on your starting point, a good first target might be to reach 20 or fewer warnings per KNCSL on the third metric, then 10, and then 5, on the way to reaching zero.

The next target is to reach zero compiler warnings in pedantic mode with all the warnings turned on—the fourth metric tallied. This is a much higher bar to meet, and therefore may also require setting a series of intermediate subgoals. Meeting this standard implies meeting a solid standard of workmanship. Many developers regard reaching this goal simply as a matter of pride. In a sense, there probably isn't much point in looking at stronger types of code checking if you can't meet this target.

Once you get this far—it could take several months—the next goal will not be too surprising: you should now reach zero warnings generated from static source code analyzers. Even though the leading static source code analyzers aren't cheap, using them routinely can make a very real difference in the quality of the code you produce.

At this point, you may be thinking that we've forgotten about the assertion count, but not to worry—it's next. We don't need to dwell much on the benefits of using assertions. Just think of an assertion as the canary in a coal mine: it can tell you at the earliest possible point in an execution that something might be off and that your code can no longer be expected to function correctly.

There's actually a statistically significant correlation between the number of assertions in code and the number of residual defects (G. Kudrjavets, N. Nagappan, and T. Ball, "Assessing the Relationship between Software Assertions and Code Quality: An Empirical Investigation," tech. report MSR-TR-2006-54, Microsoft Research, 2006). Increasing the number of assertions reduces the number of residual defects—it's that simple. As a target, we recommend reaching an assertion density of 2 to 5 percent. You do, of course, want the assertions in your code to be meaningful, so simply using "`assert(true)`" to boost the numbers won't help much.

You've now reached the end of the improvement phase, but you aren't done yet. You'll need to address one more issue here, and it's related to the adherence to a sensible coding standard. Most of the existing coding standards tend to be long and detailed, with hundreds of rules that are often only marginally related to code quality—for example, regulating the use of white space. Select a small number of coding rules, no more

METRICS FOR SUCCESS

1. Measure the number of lines of code you have.
2. Count the number of assertions in your code.
3. Measure the number of compiler warnings generated in a standard build.
4. Add -Wall and -pedantic, and again count the number of warnings.
5. Run a good static source code analyzer over the code, and count the number of warnings.
6. Use a good coding standard, and heed the rules.

than 10 or 20, that are clearly risk related, and make sure you follow them.

The best way to do this is to use a tool to check compliance. Static source code analyzers are very good at doing this. For example, you can use Codesonar or Semmle/Odasa to check compliance with the Power of Ten coding rules (http://spinroot.com/p10/) or with the coding standard that NASA's Jet Propulsion Laboratory uses for flight software development (http://lars-lab.jpl.nasa.gov/). Several analyzers can also check compliance with more bulky standards, such as the Joint Strike Fighter Coding Standard for C++ (www2.research.att.com/~bs/JSF-AV-rules.pdf).

STEP THREE: TRACK YOUR PROGRESS

Set a time schedule with specific dates for reaching each of the targets listed here. If you have a lot of work to do to reach a target, you should set subgoals, no more than three months apart, to help you track your overall progress. It's important to avoid biting off too much too soon. For example, take on the third, fourth, and fifth metrics one at a time, in that specific order, and don't try to handle the fifth metric while you're still battling the third.

You can phase in the second metric more gently, in parallel with the others: slowly but steadily increase the assertion density in your code. Pay special attention to those parts of the code that score particularly poorly on this metric. And slowly but surely, start the process of converging toward full compliance with the coding rules you selected. You chose only a small set, so you can assume that they are important, and therefore you must make every effort to follow them.

If you've reached your goals, you should see your residual defect rates decrease, and perhaps even see your boss smile just a little bit more. Notice that I haven't said anything about changes you could make in the way you test your software. Clearly, there's also likely to be room for improvement there. Although the part of the recovery process that I've described is typically ignored, it's critically important if you want to rely less on users to catch your bugs.

Acknowledgments

This research was carried out at the Jet propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Gerard Holzmann *is the lead scientist in the NASA/JPL Laboratory for Reliable Software. Contact him at gholzmann@acm.org.*

Editor: Mike Hinchey, Lero—the Irish Software Engineering Research Centre; mike.hinchey@lero.ie