Static Analysis Deployment Pitfalls

Supplemental Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering

Flash Sheridan
Code Integrity Solutions
http://pobox.com/~flash

***Abstract*—Organizational, political, and configuration mistakes in the deployment of a static source code analysis tool can eliminate most of its benefits, even while apparently meeting management goals. A list of pitfalls encountered as a static analysis consultant is presented, with discussion of techniques for avoiding or mitigating them. This article is part of a work in progress, tentatively entitled "Deploying Static Analysis."**

Keywords: static analysis, program checking, software quality assurance procedures

Introduction

In the last decade, a revolution in static source code analysis enabled its widespread deployment for bug-finding during software development. But even these new and fundamentally superior tools can lose most of their value if organizational, political, and configuration issues are not handled skillfully. The naïve approach may seem to management to be succeeding, while in the long run being nearly useless, and in the meantime bringing static analysis into disrepute with the programmers dealing with the supposed defects. I present here some of the leading pitfalls I have encountered in deploying, maintaining, and using static analysis tools within software development organizations, principally Coverity at Palm, Inc. and Klocwork as a consultant for Code Integrity Solutions.

Pitfall I: Don't Lose Focus on Finding Bugs

The primary goal of static analysis is, or at least ought to be, the improvement of code by finding and fixing defects. But finding bugs is a messy process, more familiar to software testers than to those who are usually given responsibility for static analysis: software developers or tool maintainers [1]. The imperatives for the latter two groups are generally a clearly-defined process and a smoothly running tool. These imperatives conflict with the rapid changes in configuration needed to adapt to experience in finding bugs. For decisions about finding bugs, the final say must belong to the software quality group, not development or tool maintenance [2].

The conventional wisdom recommends a management champion to provide political muscle behind static analysis, to keep the focus on fixing bugs, and to overcome the tendency to deprioritize bugs found by the tool [3]. Ship-at-All-Costs-itis is pervasive in the software industry: Quality generally has fewer constituents than the schedule.

Pitfall II: Don't Buy a Tool Based on Bugs it Finds in Other People's Code

Before you commit to a static analysis tool, make sure that it finds important bugs in your real code. Bugs found in open source or demo code can be very impressive; but your organization's code, while it's under development (which is the cheapest time to find bugs) will be very different from code which has already been made public. Also, getting a static analysis tool to work with your build system may be the hardest part of the deployment; it is essential to ensure that this will work before spending money on a tool.

Commercial static analysis vendors generally offer a free trial, and you don't have to give the bugs back if you don't buy the tool. Another upside to testing on your real code is that if the tool is right for your organization, it will sell itself, by exhibiting bugs that convince even high-ranking sceptics that it is worth its price.

Pitfall III: Don't Accept the Default Settings

Once the static analysis vendor has configured your installation, investigate and experiment with its settings on your code. The defaults are likely to be a one-size-fits-all configuration, designed to make the tool look good and minimize problems for the vendor's support staff.

The vendor may have disabled some checkers because of a high rate of falsely reported defects ("false positives" in the jargon) in rare circumstances. Conversely, some checkers may be unimpressive on your code. Some have statistical thresholds, which are probably set high to avoid false positives in mature open source code; the threshold for pre-release proprietary code may need to be set depressingly low. For example, if only a third of your developers check a function's return value, they may nonetheless be right; but the tool's default setting probably only reports a defect when a substantial majority perform this check.

I have had good luck in enabling a selection of checkers in the tool's front end which find problems similar to compiler warnings. Many of these might have been found by your compiler—but even with a strict warnings policy, some may slip through. Sometimes warnings can be suppressed from the command line; sometimes an overly-strict policy requires that all warnings be reported, with the result that no-one looks through them for the genuinely serious ones.

Pitfall IV: Don't Expect Immutable Settings

The experimentation recommended above requires considerable flexibility. Finding bugs with static analysis—like finding bugs in general—requires more adaptability, and less

stability of process, than software development. If a configuration setting is discovered to be producing excessive false positives, or missing promising defects, it should be changed immediately. A change review process which makes sense for production code can stymie bug-finding, waste engineering time on false positives, and bring the tool into disrepute with those actually using it.

Pitfall V: Don't Expect Stable Metrics

This flexibility will conflict with a natural management desire for stable metrics. If you disable a misguided checker, the number of reported defects will immediately go down; if you discover a new worthwhile checker, the number of defects will increase. This may conflict with management efforts to measure progress via the number of defects reported or processed. Dealing with this conflict requires a firm focus on the primary goal, of finding and fixing important bugs, and good judgment of quality; secondary considerations must remain secondary.

Pitfall VI: Don't Accept Broken Analyses

A static source code analysis tool (as opposed to bytecode or binary code analysis) must emulate your compiler with its metacompiler. Even a small number of discrepancies in the emulation can lead to a majority of reported defects being false positives, and (even worse) a majority of potential defects being missed ("false negatives").

Configure the tool, if possible, to stop and report failure for an analysis if a low threshold of analysis failures is exceeded; otherwise your build process should count the failure messages in the tool's log files. If there is no formal process to reject a broken analysis, a great deal of time can be wasted on flawed defects, and an unknown number of genuine bugs will be missed.

Pitfall VII: Don't Just Throw the Tool Over the Wall

My first law of static analysis [4], [5] is that developers who do static analysis voluntarily are the ones whose code needs it the least. Conversely, the developers whose code will benefit the most from static analysis will only use it if there is management backing, probably from the management champion discussed above. There are always excuses to deprioritize static analysis defects, some of them even valid [6]. Without serious management backing—in practice, not just in theory—for allocating time and personnel to investigating and fixing static analysis bugs, the tool is likely to remain shelf-ware.

Pitfall VIII: Don't Present the Defects in Random Order

Most static analysis tools present defects in essentially random order (e.g., alphabetical by checker name, or by file), which is unwise: If the first defect in a given engineer's queue is unimpressive, you may have lost him [7]. This is particularly disappointing since there are techniques in the academic literature for ranking defects by reliability, relevance, and estimated importance [7], [8]. It is particularly ironic that a co-author of these papers co-founded one of the leading static analysis vendors, which has rejected automated defect ranking [9].

Your setup and documentation should show important defects early, rather than likely false positives. I know of no general solution; one example is described in [4]. A good static analysis tool is likely to find more bugs in your code than you have time and resources to fix, so good judgment in prioritization will be crucial [10].

Pitfall IX: Don't Expect Perfect Accuracy

Any practical static analysis tool will produce false positives and don't-cares. Set expectations low enough that developers continue to evaluate defects after encountering false positives. Fixing static analysis defects will be far more efficient than conventional ones, even with a very high false positive rate. Static analysis bypasses all of testers' time, all of an engineer's bug isolation/location time, and usually almost all analysis time: The tool goes right to the heart of the defect, and highlights the problem in clear, bright colors.

Pitfall X: Don't Assign Your Most Junior Programmers to Static Analysis Defects

Don't assign your least skilled programmers to classify and fix defects reported by static analysis. Evaluating these is two levels of abstraction higher than writing code: Finding bugs in your code *should* be hard, and evaluating potential mistakes in such alleged bugs is harder still.

Even more importantly, the most valuable benefit from static analysis, greater even than fixing bugs, is preventing future bugs, by educating the developer about his or her mistakes. If someone else is looking at these bugs, the developer never sees the mistakes and cannot learn from them.

References

[1] Andy Yang and Flash Sheridan, "Why testers should care about static analysis," *The Testing Planet,* July 2010, 16-7.

[2] Joel Spolsky, *Joel on Software,* Apress 2004.

[3] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, volume 53, number 2 (2010), pp. 66-75.

[4] Flash Sheridan, "Static analysis in a fallen world," talk at Stanford University Computer Science Department, 15 January 2010.

[5] Andy Yang, "The benefits of source code analysis," Code Integrity Solutions white paper.

[6] Bill Pugh, "Cost of static analysis for defect detection," talk at Stanford University Computer Science Department, 20 April 2009.

[7] Ted Kremenek and Dawson Engler, "Z-ranking: using statistical analysis to counter the impact of static analysis approximations," *Proceedings of the 10th Annual International Static Analysis Symposium,* Springer-Verlag 2003, pp. 295-315.

[8] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler, "Correlation exploitation in error ranking," *SIGSOFT Software Engineering Notes* 29, 6 (Nov. 2004), pp. 83-93.

[9] Dawson Engler, personal communication, 11 May 2009.

[10] Flash Sheridan, "Handling an embarrassment of riches," Code Integrity Solutions blog posting, 29 January 2010.