Holy Triage, Batman!

Code triage: it's a dirty job, but somebody's got to do it—quickly, well and often on very little notice.

PREVIOUS
Shawn Powers' The Open-Source Classroom

NEXT
New Products

SUSAN SONS

Susan Sons serves as a Senior Systems Analyst at Indiana University's Center for Applied Cybersecurity Research (http://cacr.iu.edu), where she divides her time between helping NSF-funded science and infrastructure projects improve their security, helping secure a DHS-funded static analysis project, and various attempts to save the world from poor information security practices in general. Susan also volunteers as Director of the Internet Civil Engineering Institute (http://icei.org), a nonprofit dedicated to supporting and securing the common software infrastructure on which we all depend. In her free time, she raises an amazing mini-hacker, writes, codes, researches, practices martial arts, lifts heavy things and volunteers as a search-and-rescue and disaster relief worker.

ALMOST ONCE PER WEEK, SOMEONE COMES TO ME ASKING FOR A CODE AUDIT. Almost invariably, these people have no idea what they are asking for. In 99% of cases, what they really need is *code triage*, or perhaps a more in-depth review, not an audit.

A real code audit means *auditing to a standard*, a process in which code is checked to see that it complies fully with a specific, objective coding standard. It can be done, but it's resource-intensive for all but the most trivial of cases. Unless the standard is unusually well suited to ensuring something very much needed by the code's use case, this type of audit is usually more socially useful than it is technologically useful.

More commonly, what people requesting a code audit mean is *make me a one-page list of every way in which this software deviates from the ideal*. There are

a number of things wrong with this:

1. You probably can't provide a specific and extensive enough definition of correct behavior for your software, taking into account all possible inputs, environments and eventualities, to provide a standard to evaluate its total correctness. To audit is to *compare code to an objective standard*, without such a standard, audit is impossible.

2. I cannot find *every* problem in your code while you are still developing it, and you probably aren't willing to undergo a one- to two-year code freeze while your code is analyzed.

3. You probably aren't willing to pay for one to four man-years of senior software security engineer time to come close to finding *every* potential problem with your code, which is typical for a medium-complexity project with mediocre code quality and a few hundred kloc.

I do sometimes say yes to *code reviews*, and I often find myself doing *code triage*. What's the difference?

Code review is systematic examination of computer source code intended to find mistakes overlooked in the initial development phase, improving the overall quality of software.

Code triage is a specific form of code review intended to identify the most critical targets for immediate improvement without a deep inspection. In other words, code triage answers the question, "Given limited time and resources, how do I address this code's worst deficiencies?"

The Role of Triage

One of the scariest moments in a developer's life—especially someone who works with infrastructure software or anything security-critical—is taking over someone else's mess. Even someone else's good code—in enough volume, on tight enough deadlines, with little enough documentation, tooling and familiarity—can seem like a mess. Code triage is the method for making sense of the mess, instead of saying "I can't touch this until I've had 12 solid man-months to spelunk its depths."

Let's assume you don't have 12 man-months.

Let's pretend you just discovered that a critical piece of infrastructure software has gone unmaintained, or badly mismanaged, for about a decade. You don't know how bad it is, but you suspect it is terrible. It's the kind of software that, if it breaks just so, people will die, or the world economy will be in ruins—whatever. This has somehow (through assignment or your unrelenting sense of duty) become your problem.

This may or may not have happened to me before. I can promise you, there's a way to win through, and once you've become practiced at it, having a few hundred-thousand lines of broken mystery code fall in your lap will become significantly less daunting. Note that I didn't say "sane" or "easy"—just "less daunting". There is method to the madness.

The People Phase

Never start with the code if you can help it. The code will tell you what someone programmed, but it never will tell you what someone intended to program or, for that matter, what they *should have* intended to program. I begin code triage on complex projects (anything that seems to deserve more than a half-day of my time) with a fountain pen and a notebook. Vary the tools if you must, but the process is important.

Keep excellent notes. Remember that triage is a process of gathering information to use in a decision-making process. If information is lost before it can be used, you've failed. Triage is often undertaken as a one-person activity, but it may be done by a small team if the team is tightly knit and well coordinated enough. If working with a team, notes must be kept in such a way that team members can collaborate in as close to real time as possible, and that the notes are clear to all of the team at all times. Getting to that point with a team is difficult but possible.

Your complete notes should be close at hand at all times during the triage process. Often, information you come across in one part of the process will conflict with, or relate to, something that came up earlier. These relationships are typically the most important discoveries and may escape notice if notes are disjointed or compiled only after the fact. Keep it together! Review the entire body of notes often.

Use varied sources. You will want to contact as many different stakeholders as possible, including at least one sample from any distinct

What problems have others identified already? What barriers have others found when they have tried to make improvements? What frustrates people? What changes are people afraid of? What functionality do they depend on?

group of stakeholders you can identify. For example, when triaging the NTP reference implementation, I spoke to the project's maintainer, to the project's funding coordinator, to community members who had contributed code or tried and failed to contribute code, to past contributors who had left the project, to a package maintainer who packaged the software for a major Linux distribution, and to end users in different sectors where the software was employed: commodity computer usage, data center, core internet routing, scientific applications and finance sector. I did not have the opportunity to talk to the software's original author, but I made an effort to get to know a bit about him through his writing.

Listen mindfully. When interviewing people as part of software triage, expect to get different perspectives from different people. Don't try to get consensus; it's not important at this stage. You are gathering information and not much else. What problems have others identified already? What barriers have others found when they have tried to make improvements? What frustrates people? What changes are people afraid of? What functionality do they depend on?

I tend to ask users most about their use cases. I ask developers most about developer experience, the purpose of the software, and so on. And, I ask everyone about the things they would like to change and not like to change. However, the most important thing that I look for is something that, by nature, I cannot ask:

What assumptions do I hear when people talk about the software?

Unspoken assumptions can be dangerous, and until I've gone through the documentation in detail, I do not know which assumptions are or are not explicitly documented. So, I assume all assumptions have been left

unspoken until proven otherwise. Document assumptions regardless of their correctness. Spotting them takes practice.

"What time would you like to go to dinner?" assumes that you would like to go to dinner, and that you have an opinion about what time you would like to go.

"I set my pencil on the desk a moment ago; if it isn't still there, it must have rolled off" assumes that no one has picked up the pencil, that the pencil isn't capable of walking or flying, that the pencil is capable of rolling, that pencils don't evaporate, and that nothing on your desk ate the pencil.

Try This Exercise: Someone says, "This program tells you how many files are in your home directory." What are some questions you could ask about such a simple program to root out unspoken assumptions about the program's expected behavior? See the sidebar at the end of this article for a list of possible questions.

Put others at ease. This is the most difficult part, especially for many software engineers for whom interviewing is not part of their core skill set. If you are new at this or unsure of yourself, start with the consumers of the software: they are the easiest, because you often can deflect from issues of the software itself by focusing on their workflow and use case, and they usually don't see themselves as responsible for the current state of the software. If they aren't very technical, get the workflow and use-case information from them directly, then see if one of the consumers can get you in touch with someone in their IT department who supports the software for them: that's an important stakeholder too, with potentially crucial information on factors such as operating environment.

I tend to start with the current or former developer(s) of a project I'm trying to triage. Regardless of whether I'm triaging in order to take over or triaging in order to assist the standing team, these will be the most delicate interviews. They also are potentially the most fruitful. Who better knows the assumptions with which the code has been developed so far than the people developing it? Who knows how it got to where it is today? Who knows what users or potential users, contributors or potential contributors ask the development team the most? Who knows where the tooling is falling down or a struggle to work with? Who has the most ego involved with the current state of

the software? Yep. Be prepared to tread lightly.

First and foremost: *do not take an adversarial mindset into any interview*, no matter how badly you think anyone may have done his or her job. It will come through in your speech and behavior, and it will make people shut down on you in self defense. Your job is to fix the software, and you will be most effective at that if you can find some empathy for the people with whom you are speaking. This is another reason I start with the people before the code. I want to approach those people with empathy and understanding, but I'm still a cynical, grumpy engineer—at least on the days when I've been slogging through 200,000 lines of code trying to find the race condition that ate Manhattan. Don't meet people the day you wrestled with their bad code.

I keep interviews about software triage as informal as possible. Formality causes most people to expect an adversarial experience, which is exactly what I do not want to provide. I want to make people feel comfortable. Usually, this means dropping a short, informal email to set up a time convenient to them and then doing a face-to-face meeting (if possible) over tea or coffee, or a chat by phone or video conference. If face to face, make the effort to incorporate snacks/beverages: eating and drinking is a natural signal to the body that we are not in combat, and it may have a calming effect.

Bad code is bad for a reason. Let the developers tell their side of the story. Sometimes it's self-inflicted, sometimes it's not. You can judge later. Your job during the interview is to be empathetic and let them talk. Nobody gets up in the morning and decides to write terrible code just for fun. It tends to result from developers working in a toxic environment, or being in over their heads, or lacking resources, or being burnt out or some horrible dysfunction—something went wrong. Chances are, the developers can't or won't identify this and tell you directly, but if you get them talking about the software long enough, let them ramble a bit, and ask pointed questions here and there in an empathetic manner, you will get it eventually. What is most important will vary a great deal from case to case, so this stage takes patience; follow threads and see where they go.

Assume that people do not want their comments and reflections attributed to them unless they have specifically given you permission to

quote them. Fears of causing drama or feeding a rumor mill will cause people to self-censor.

In one case, I found that the most relevant tale was when the software lost a major funding source and the team ended up adding features that didn't fit their vision for the project in order to keep funding flowing from other sources, rather than let the software die. This was very helpful information, because it told me I needed to understand the project's current funding strategies when I made triage recommendations.

In another, the project lead retired and hadn't planned well enough for the project's longevity. Not having another plan, he handed the software off to his former assistant who wasn't ready for the responsibility. Years of being in over his head made the assistant a paranoid and dysfunctional project lead who chased off developers and drove the software into crisis. I burned a great deal of time and energy trying to get this project lead to cooperate with saving his software and never did succeed. I don't regret trying. Over many long, late-night phone calls, I got an inside view of how he'd struggled to balance the interests of those he saw as his most important stakeholders. Most of his views on managing the code were off enough not to be very helpful, but coming to understand how he'd gone about interacting with people made a big difference, even when he hadn't been interacting effectively.

In yet another case, the software had been written by company A to manage a specific hardware platform purchased by several companies, including B. B became so dependent on the software that when A stopped maintaining the software, B made a deal to buy it. B, however, wasn't a software development firm and had no in-house resources for software development, so it hired a series of contractors for one-off improvements or feature-adds to the software. After eight years and more than 20 contractors, the software was a security and reliability nightmare with no design integrity whatsoever, no documentation and a brittle build system. When I got to the code, I was prepared to deal with the huge variation in coding styles I found and the lack of design integrity. I also was able to get in touch with someone in the accounting department who could help me reconstruct which contractors I would want to speak with based on dates of various code changes, and I actually was able reach some of them.

Reliable, reproducible builds are necessary to the development process, and little improvement to the code can happen without them.

Don't be afraid to go back. Often, after talking to more people or during a later stage of the triage process, you will trip on something that leaves you wanting to speak with someone you've already spoken to. I end all interviews by asking permission to contact the people again should such a need arise and ask how they prefer to be contacted so that I may be as respectful of their time as possible.

The Proxy Phase

Before I jump headlong into a big code base, but after I've spoken to whatever relevant people I can reach, I still have work to do around the code. I spend some time looking for things that can give me red flags of likely problems, or signs that certain other things may be well handled, or tools that may exist to make my work with the code easier. I call this the *proxy stage* because many of the things I look at aren't actually direct evidence of what's broken; they're just strongly correlated enough to be useful in a quick triage.

The proxy stage will direct the effort you put in for the rest of your triage process. It's triage for triage. When you find out that the software cannot be built without the one machine in a former developer's apartment that no one has root on, which has a black-box script no one knows the contents of, the build system becomes a priority far higher than the contents of the code base. Reliable, reproducible builds are necessary to the development process, and little improvement to the code can happen without them.

On the other hand, discovering that the source control, build system and so on are in good shape tells you that spending time digging through SCM logs probably will give you useful information, because someone took the time to use those tools properly.

Begin with documentation. Hopefully you already know how useful good documentation can be; however, don't discount the potential

treasures to be found in bad documentation:

- Who authored what parts of the documentation can tell you about who cares about what components, other than the developers.
- Relative ages of different parts of the documentation can give you an idea of the relative neglect of different parts of the code, in the absence of revision control or other, better data.
- Insight into the mental models that developers were operating on while writing code.

Spelunk issue queue contents (current and historical).

- Find out how issues have been handled in the past; this will tell you a lot about the development team's workflow.
- Find out what big issues have been churned on for a long time but not solved.
- Find out what security issues have cropped up in the past and how they were dealt with.
- Find out how active the community/team is in general.

Look at tests.

- Are there tests?
- What is the coverage like?
- What is the overall sophistication like? For example, is it a unit-test-only setup, a functional-test-only setup, or are both in use? Has this project begun using fuzz testing? Is there scaffolding for mocked interfaces?
- Do all tests currently run and pass?

- Can you tell anything about the testing strategies in use? Were tests committed with every patch? Was adversarial testing employed? Was only one person writing tests? Was anyone a testing specialist?

Examine tooling (build system, CI infrastructure, source control setup and so on).

- How much automation is/was in place?
- How reliable is the automation? How much is still available/usable?
- Is a modern SCM (git or Mercurial) in use?
- Do the tools seem to be reducing the dev team's overhead or increasing it? (That is, is it doing its job, which is making developers' lives easier and their work better?)

Use commit messages, tags and branch structure within the SCM.

- Are commit messages, tags and branching used effectively—that is, can you follow them?
- What can you learn from reading the commit messages?

Look at general style and code quality.

- Don't get sucked into a deep read of the code yet, skim only.
- How is the overall comment density? Are the comments literate?
- Does the indentation, overall structure and so on suggest the absence or presence of a style guide?
- Does this feel "clean" or "messy" in general?
- Is semantic versioning used?

- How many red flag comments do you see in a quick skim? Look for things that indicate cut-and-paste coding—for example, "got this from <url>" or anything mentioning Stack Overflow. Also, look for "WTF", "IDK why, but if I remove this, it breaks", and things like that. These are areas you may want to look at in the code stage if you have time.

The Code Phase

You may have run out of time for triage by now, or you may have found so many problems that triage is done. I have had projects like that. When I stepped into NTP, the code wasn't C99-compliant, the build system was unusable, the code was locked up in an inaccessible and antiquated SCM, and the documentation was mostly more than seven years out of date—all of those issues took precedence over specific code improvements, because fixing them was a prerequisite to enabling developers to fix the code. We needed to be able to onboard new developers by giving them access to code they could actually build and documentation on how it all worked.

Do not try to read, let alone understand, all or most of the code. Your job is not to find every problem—90% of problems are irrelevant to your search, unless the code is shockingly well written. Remember, you are doing triage: you are a field medic, you are not performing an autopsy. You are figuring out how to do the most to improve the life of the patient in limited time with limited resources: where do I get the most bang for my buck *right now*, and what do I look at next once that is done? Nothing more. Your proxy stage, above, will give you a clue about how much of the code stage to bother with. In most cases, you will skip some or most of it.

You will get your biggest gains by improving development process (because then fixing code becomes faster/easier, and all development after that point gives greater returns), by fixing extremely high-impact vulnerabilities and by making changes that remove entire classes of vulnerabilities (rather than trying to squash them one at a time). To that end, see below.

Evaluate program architecture. Think about the code's overall architecture. Are you having trouble navigating it? How good or bad is

the separation of concerns? How well is minimization being practiced? Don't spend too much time trying to learn it all, just skim for major red flags—for example, crypto algorithms housed in the same place as web interface code or piles of theoretically unreachable code.

Eliminating code eliminates attack surface, eliminating entire classes of vulnerabilities. It also reduces complexity, reducing opportunities for developer mistakes. If you can safely remove code, do so.

Refactoring to make code more navigable and more logically compartmentalized makes it easier for developers to understand, makes bug fixing easier, reduces the rate of defects introduced by developer error and increases the speed at which developers can introduce high-quality, atomic patches. It isn't always highest priority in a disastrous code base though, as brittle code bases are difficult and time-consuming to refactor. Other changes likely will take priority if the code resembles spaghetti.

Catalog interfaces. Find and list all of the software's internal and external interfaces—or all of the ones you can find. Try to figure out which ones are well defined and controlled, and which aren't, and figure out which are used and necessary, and which aren't.

Catalog data stores and data sources. Most software deals with data at some point. Look at where external data comes from, what assumptions are made and how the software copes with nonconforming (accidentally or maliciously) or missing data. Now do it again for any data the software stores.

Remain mindful of assumptions. As you go through the code, keep in mind all the assumptions you noted earlier. Note anything in the code that confirms or conflicts with those assumptions. Note any new assumptions you find.

Putting It All Together

Don't go down rabbit holes. A good software engineer will be tempted, at some point, to dive in to an interesting problem. Six hours later, your triage will be shot. You *do not* have time to understand any one problem fully; you are trying to understand the breadth of the problems the software has. This is not an exercise in depth. Don't be afraid to make generalizations and intuitive leaps, as long as you note

them as such and jot down a rough estimate of the time it would take to investigate fully the issues involved.

Triage is more complicated when you do not understand the problem space, but that complication largely can be conquered by carefully compartmentalizing the problem. Be methodical, and don't get distracted by the esoteric bits of domain knowledge you don't have. If the software is not properly segmented so that very domain-specific algorithms are separate from interfaces, crypto and so on, that is a fault in itself. Note it, and move on. If it is well segmented, you should have no problem checking out the build system, data stores, interfaces and so on, leaving the domain-specific code segments for deep dives with a domain expert by your side.

The first thing you will want to do is to use the information you just gathered to aid in decision-making and communicate that process to others. Lay out a plan, and describe what led you to choose that plan. Keep your notes, and ensure that the references to specific code in those notes will be find-able later, after the code has evolved and/or been moved to another SCM.

If you can, follow the software through at least the first stages of its refactor or rejuvenation following the triage you just did. That experience provides a crucial feedback loop that will enable you to improve your triage skills much faster than you could without it. You'll inevitably see things you missed: some that you had no chance of finding without a deep dive, and some that you'll soon realize were staring you in the face all along. The more of these experiences you have, the better you will become at triage.

Practice is *the* way to improve your code triage skills. Good practice is frequent and in volume. Long breaks make it hard to build on and reinforce previous learning. Working with only small code samples will not teach you the skills needed to find big-picture problems and trends in a sea that you cannot read line by line. Additionally, try to work with a variety of code, in terms of language, domain and quality. If you can, also stretch your assessment muscles in other domains. I've learned many triage skills in volunteer search-and-rescue work that transferred to software engineering and information security.■

SOME ANSWERS TO THE "TRY THIS EXERCISE"

- How does it determine the scope of "my home directory"?
- Will it span multiple devices?
- Will it follow symlinks?
- If the same file is symlinked or hardlinked many times, how will it be counted?
- What if there is a hardlink or symlink in my home directory to something outside my home directory?
- Is any type of deduplication attempted?
- What if I, for some reason, had filesystem objects in my home directory that aren't really files per se, such as broken filesystem pointers or half of /proc?
- How does the program determine what my home directory is?
- If `$HOME` differs from what is listed as my home in /etc/passwd will that have any effect?

Send comments or feedback via http://www.linuxjournal.com/contact or to ljeditor@linuxjournal.com.

RETURN TO CONTENTS