How to Report Bugs Effectively

by Simon Tatham, professional and free-software programmer

Introduction

Anybody who has written software for public use will probably have received at least one bad bug report. Reports that say nothing ("It doesn't work!"); reports that make no sense; reports that don't give enough information; reports that give *wrong* information. Reports of problems that turn out to be user error; reports of problems that turn out to be the fault of somebody else's program; reports of problems that turn out to be network failures.

There's a reason why technical support is seen as a horrible job to be in, and that reason is bad bug reports. However, not all bug reports are unpleasant: I maintain free software, when I'm not earning my living, and sometimes I receive wonderfully clear, helpful, *informative* bug reports.

In this essay I'll try to state clearly what makes a good bug report. Ideally I would like everybody in the world to read this essay before reporting any bugs to anybody. Certainly I would like everybody who reports bugs to *me* to have read it.

In a nutshell, the aim of a bug report is to enable the programmer to see the program failing in front of them. You can either show them in person, or give them careful and detailed instructions on how to make it fail. If they can make it fail, they will try to gather extra information until they know the cause. If they can't make it fail, they will have to ask you to gather that information for them.

In bug reports, try to make very clear what are actual facts ("I was at the computer and this happened") and what are speculations ("I *think* the problem might be this"). Leave out speculations if you want to, but don't leave out facts.

When you report a bug, you are doing so because you want the bug fixed. There is no point in swearing at the programmer or being deliberately unhelpful: it may be their fault and your problem, and you might be right to be angry with them, but the bug will get fixed faster if you help them by supplying all the information they need. Remember also that if the program is free, then the author is providing it out of kindness, so if too many people are rude to them then they may stop feeling kind.

"It doesn't work."

Give the programmer some credit for basic intelligence: if the program really didn't work at all, they would probably have noticed. Since they haven't noticed, it must be working for them. Therefore, either you are doing something differently from them, or your environment is different from theirs. They need information; providing this information is the purpose of a bug report. More information is almost always better than less.

Many programs, particularly free ones, publish their list of known bugs. If you can find a list of known bugs, it's worth reading it to see if the bug you've just found is already known or not. If it's already known, it probably isn't worth reporting again, but if you think you have more information than the report in the bug list, you might want to contact the programmer anyway. They might be able to fix the bug more easily if you can give them information they didn't already have.

This essay is full of guidelines. None of them is an absolute rule. Particular programmers have particular ways they like bugs to be reported. If the program comes with its own set of bug-reporting guidelines, read them. If the guidelines that come with the program contradict the guidelines in this essay, follow the ones that come with the program!

If you are not reporting a bug but just asking for help using the program, you should state where you have already looked for the answer to your question. ("I looked in chapter 4 and section 5.2 but couldn't find anything that told me if this is possible.") This will let the programmer know where people will expect to find the answer, so they can make the documentation easier to use.

"Show me."

One of the very best ways you can report a bug is by showing it to the programmer. Stand them in front of your computer, fire up their software, and demonstrate the thing that goes wrong. Let them watch you start the machine, watch you run the software, watch how you interact with the software, and watch what the software does in response to your inputs.

They know that software like the back of their hand. They know which parts they trust, and they know which parts are likely to have faults. They know intuitively what to watch for. By the time the software does something obviously wrong, they may well have already noticed something *subtly* wrong earlier which might give them a clue. They can observe everything the computer does during the test run, and they can pick out the important bits for themselves.

This may not be enough. They may decide they need more information, and ask you to show them the same thing again. They may ask you to talk them through the procedure, so that they can reproduce the bug for themselves as many times as they want. They might try varying the procedure a few times, to see whether the problem occurs in only one case or in a family of related cases. If you're unlucky, they may need to sit down for a couple of hours with a set of development tools and *really* start investigating. But the most important thing is to have the programmer looking at the computer when it goes wrong. Once they can see the problem happening, they can usually take it from there and start trying to fix it.

"Show me how to show myself."

This is the era of the Internet. This is the era of worldwide communication. This is the era in which I can send my software to somebody in Russia at the touch of a button, and he can send me comments about it just as easily. But if he has a problem with my program, he *can't* have me standing in front of it while it fails. "Show me" is good when you can, but often you can't.

If you have to report a bug to a programmer who can't be present in person, the aim of the exercise is to enable them to *reproduce* the problem. You want the programmer to run their own copy of the program, do the same things to it, and make it fail in the same way. When they can see the problem happening in front of their eyes, then they can deal with it.

So tell them exactly what you did. If it's a graphical program, tell them which buttons you pressed and what order you pressed them in. If it's a program you run by typing a command, show them precisely what command you typed. Wherever possible, you should provide a verbatim transcript of the session, showing what commands you typed and what the computer output in response.

Give the programmer all the input you can think of. If the program reads from a file, you will probably need to send a copy of the file. If the program talks to another computer over a network, you probably can't send a copy of that computer, but you can at least say what kind of computer it is, and (if you can) what software is running on it.

"Works for me. So what goes wrong?"

If you give the programmer a long list of inputs and actions, and they fire up their own copy of the program and nothing goes wrong, then you haven't given them enough information. Possibly the fault doesn't show up on every computer; your system and theirs may differ in some way. Possibly you have misunderstood what the program is supposed to do, and you are both looking at exactly the same display but you think it's wrong and they know it's right.

So also describe what happened. Tell them exactly what you saw. Tell them why you think what you saw is wrong; better still, tell them exactly what you expected to see. If you say "and then it went wrong", you have left out some very important information.

If you saw error messages then tell the programmer, carefully and precisely, what they were. They *are* important! At this stage, the programmer is not trying to fix the problem: they're just trying to find it. They need to know what has gone wrong, and those error messages are the computer's best effort to tell you that. Write the errors down if you have no other easy way to remember them, but it's not worth reporting that the program generated an error unless you can also report what the error message was.

In particular, if the error message has numbers in it, *do* let the programmer have those numbers. Just because you can't see any meaning in them doesn't mean there isn't any. Numbers contain all kinds of information that can be read by programmers, and they are likely to contain vital clues.

Numbers in error messages are there because the computer is too confused to report the error in words, but is doing the best it can to get the important information to you somehow.

At this stage, the programmer is effectively doing detective work. They don't know what's happened, and they can't get close enough to watch it happening for themselves, so they are searching for clues that might give it away. Error messages, incomprehensible strings of numbers, and even unexplained delays are all just as important as fingerprints at the scene of a crime. Keep them!

If you are using Unix, the program may have produced a core dump. Core dumps are a particularly good source of clues, so don't throw them away. On the other hand, most programmers don't like to receive huge core files by e-mail without warning, so ask before mailing one to anybody. Also, be aware that the core file contains a record of the complete state of the program: any "secrets" involved (maybe the program was handling a personal message, or dealing with confidential data) may be contained in the core file.

"So then I tried . . ."

There are a lot of things you might do when an error or bug comes up. Many of them make the problem worse. A friend of mine at school deleted all her Word documents by mistake, and before calling in any expert help, she tried reinstalling Word, and then she tried running Defrag. Neither of these helped recover her files, and between them they scrambled her disk to the extent that no Undelete program in the world would have been able to recover anything. If she'd only left it alone, she might have had a chance.

Users like this are like a mongoose backed into a corner: with its back to the wall and seeing certain death staring it in the face, it attacks frantically, because doing something has to be better than doing nothing. This is not well adapted to the type of problems computers produce.

Instead of being a mongoose, be an antelope. When an antelope is confronted with something unexpected or frightening, it freezes. It stays absolutely still and tries not to attract any attention, while it stops and thinks and works out the best thing to do. (If antelopes had a technical support line, it would be telephoning it at this point.) Then, once it has decided what the safest thing to do is, it does it.

When something goes wrong, immediately stop doing *anything*. Don't touch any buttons at all. Look at the screen and notice everything out of the ordinary, and remember it or write it down. Then perhaps start cautiously pressing "OK" or "Cancel", whichever seems safest. Try to develop a reflex reaction - if a computer does anything unexpected, freeze.

If you manage to get out of the problem, whether by closing down the affected program or by rebooting the computer, a good thing to do is to try to make it happen again. Programmers like problems that they can reproduce more than once. Happy programmers fix bugs faster and more efficiently.

"I think the tachyon modulation must be wrongly polarised."

It isn't only non-programmers who produce bad bug reports. Some of the worst bug reports I've ever seen come from programmers, and even from good programmers.

I worked with another programmer once, who kept finding bugs in his own code and trying to fix them. Every so often he'd hit a bug he couldn't solve, and he'd call me over to help. "What's gone wrong?" I'd ask. He would reply by telling me his current opinion of what needed to be fixed.

This worked fine when his current opinion was right. It meant he'd already done half the work and we were able to finish the job together. It was efficient and useful.

But quite often he was wrong. We would work for some time trying to figure out why some particular part of the program was producing incorrect data, and eventually we would discover that it wasn't, that we'd been investigating a perfectly good piece of code for half an hour, and that the actual problem was somewhere else.

I'm sure he wouldn't do that to a doctor. "Doctor, I need a prescription for Hydroyoyodyne." People know not to say that to a doctor: you describe the symptoms, the actual discomforts and aches and pains and rashes and fevers, and you let the doctor do the diagnosis of what the problem is and what to do about it. Otherwise the doctor dismisses you as a hypochondriac or crackpot, and quite rightly so.

It's the same with programmers. Providing your own diagnosis might be helpful sometimes, but always state the symptoms. The diagnosis is an optional extra, and not an alternative to giving the symptoms. Equally, sending a modification to the code to fix the problem is a useful addition to a bug report but not an adequate substitute for one.

If a programmer asks you for extra information, don't make it up! Somebody reported a bug to me once, and I asked him to try a command that I knew wouldn't work. The reason I asked him to try it was that I wanted to know which of two different error messages it would give. Knowing which error message came back would give a vital clue. But he didn't actually try it - he just mailed me back and said "No, that won't work". It took me some time to persuade him to try it for real.

Using your intelligence to help the programmer is fine. Even if your deductions are wrong, the programmer should be grateful that you at least *tried* to make their life easier. But report the symptoms as well, or you may well make their life much more difficult instead.

"That's funny, it did it a moment ago."

Say "intermittent fault" to any programmer and watch their face fall. The easy problems are the ones where performing a simple sequence of actions will cause the failure to occur. The programmer can then repeat those actions under closely observed test conditions and watch what happens in great detail. Too many problems simply don't work that way: there will be programs that fail once a week, or fail once in a blue moon, or never fail when you try them in front of the programmer but always fail when you have a deadline coming up.

Most intermittent faults are not truly intermittent. Most of them have some logic somewhere. Some might occur when the machine is running out of memory, some might occur when another program tries to modify a critical file at the wrong moment, and some might occur only in the first half of every hour! (I've actually seen one of these.)

Also, if you can reproduce the bug but the programmer can't, it could very well be that their computer and your computer are different in some way and this difference is causing the problem. I had a program once whose window curled up into a little ball in the top left corner of the screen, and sat there and *sulked*. But it only did it on 800x600 screens; it was fine on my 1024x768 monitor.

The programmer will want to know anything you can find out about the problem. Try it on another machine, perhaps. Try it twice or three times and see how often it fails. If it goes wrong when you're doing serious work but not when you're trying to demonstrate it, it might be long running times or large files that make it fall over. Try to remember as much detail as you can about what you were doing to it when it did fall over, and if you see any patterns, mention them. Anything you can provide has to be some help. Even if it's only probabilistic (such as "it tends to crash more often when Emacs is running"), it might not provide direct clues to the cause of the problem, but it might help the programmer reproduce it.

Most importantly, the programmer will want to be sure of whether they're dealing with a true intermittent fault or a machine-specific fault. They will want to know lots of details about your computer, so they can work out how it differs from theirs. A lot of these details will depend on the particular program, but one thing you should definitely be ready to provide is version numbers. The version number of the program itself, and the version number of the operating system, and probably the version numbers of any other programs that are involved in the problem.

"So I loaded the disk on to my Windows . . ."

Writing clearly is essential in a bug report. If the programmer can't tell what you meant, you might as well not have said anything.

I get bug reports from all around the world. Many of them are from non-native English speakers, and a lot of those apologise for their poor English. In general, the bug reports with apologies for their poor English are actually very clear and useful. All the most unclear reports come from native English speakers who assume that I will understand them even if they don't make any effort to be clear or precise.

- *Be specific*. If you can do the same thing two different ways, state which one you used. "I selected Load" might mean "I clicked on Load" or "I pressed Alt-L". Say which you did. Sometimes it matters.
- *Be verbose*. Give more information rather than less. If you say too much, the programmer can ignore some of it. If you say too little, they have to come back and ask more questions. One bug report I received was a single sentence; every time I asked for more information, the reporter would reply with another single sentence. It took me several weeks to get a useful amount of information, because it turned up one short sentence at a time.
- *Be careful of pronouns*. Don't use words like "it", or references like "the window", when it's unclear what they mean. Consider this: "I started FooApp. It put up a warning window. I tried to close it and it crashed." It isn't clear what the user tried to close. Did they try to close the warning window, or the whole of FooApp? It makes a difference. Instead, you could say "I started FooApp, which put up a warning window. I tried to close the warning window, and FooApp crashed." This is longer and more repetitive, but also clearer and less easy to misunderstand.
- *Read what you wrote*. Read the report back to yourself, and see if *you* think it's clear. If you have listed a sequence of actions which should produce the failure, try following them yourself, to see if you missed a step.

Summary

- The first aim of a bug report is to let the programmer see the failure with their own eyes. If you can't be with them to make it fail in front of them, give them detailed instructions so that they can make it fail for themselves.
- In case the first aim doesn't succeed, and the programmer *can't* see it failing themselves, the second aim of a bug report is to describe what went wrong. Describe everything in detail. State what you saw, and also state what you expected to see. Write down the error messages, *especially* if they have numbers in.
- When your computer does something unexpected, *freeze*. Do nothing until you're calm, and don't do anything that you think might be dangerous.
- By all means try to diagnose the fault yourself if you think you can, but if you do, you should still report the symptoms as well.
- Be ready to provide extra information if the programmer needs it. If they didn't need it, they wouldn't be asking for it. They aren't being deliberately awkward. Have version numbers at your fingertips, because they will probably be needed.
- Write clearly. Say what you mean, and make sure it can't be misinterpreted.
- Above all, *be precise*. Programmers like precision.

Disclaimer: I've never actually seen a mongoose or an antelope. My zoology may be inaccurate.