# How to Make your Bugs Lonely
# Tips on Bug Isolation

**Danny R. Faught**

**Tejas Software Consulting**

**faught@tejasconsulting.com**

**Bio:** Danny R. Faught frequently writes and speaks about software testing, including papers and courses that cover bug isolation. He is the Defect Tracking Technical Advisor for StickyMinds.com. Danny has 11 years of testing experience.

**Abstract:** Yikes! Testers often trip across software bugs. The more we can isolate the problems we encounter, the more likely it is that we can get the problems fixed. From the first symptom to reporting and then debugging, we go through a process of isolation to zero in on where the bug lies. Bug isolation is a highly creative process, but also one that requires a great deal of discipline. Sometimes this takes many hours of effort. Have you thought about making that process more efficient? This paper will highlight practices that help to efficiently narrow down the possible causes of a bug. It will discuss the top down vs. bottom up approach and which one often send us in the wrong direction. There will be examples of using the bisection technique to quickly narrow down a precise quantitative value. There is a gray area between the defect isolation performed by a tester and the debugging that the programmer does. Finally, we'll explore the pros and cons of the different ways to split this task between the two roles.

## Introduction

*“A problem well-stated is half-solved.”* –Charles Kettering

Bug reporting starts with a symptom—something in the software doesn't work right. Some testers would report that symptom and move on. After all, there's a lot more testing to be done, right? But most testers could do much more to isolate the bug before they report it, and their organization would likely be better off for it. In this paper, I will give you some tips on how you can discover additional details about the bugs you find, and thus write more specific bug reports that are more likely to be result in getting the bugs fixed.

I really like the quote “A problem well-stated is half-solved.” It captures the fact that reporting a bug can be fully half of the process of fixing it, perhaps more. But only if the tester takes the initiative to do the work to isolate the bug, so that the bug report is an array of spotlights pointing to a very lonely bug with very little cover to hide behind.

I took a couple of shortcuts to simplify the wording in this paper, assuming that you're using a bug tracking database, assuming that you might be testing a client/server system rather than a simple desktop application, and using terms related to a GUI or web-based interface. If your situation is different, the concepts presented here should still apply. Also, I use the term “bug” where others may say “defect” to mean the same thing.

## What happened?

Your quest to isolate a software bug starts with the “Oops!” You notice some symptom that indicates that something might not be right.

At first, treat each bug as if it were a rare occurrence, and carefully collect any data that you can. Get into a habit of never dismissing an error dialog without considering whether you should capture it. Save the exact text of any error messages you see, and take screen shots. If you get a string of errors, save all of them. Save as much as you can – disk space is cheap, and it only takes a few moments to copy and paste text into a file, or to dump the state of the screen. Have a favorite text editor and image editing software close at hand.

Why take such care? Sometimes you really might not be able to reproduce the bug. The evidence that you collect at the scene of the first symptom may be the only data that you're able to report. Having the details exactly right is far better than trying to remember what an error message said. Also, you might get slightly different symptoms each time. By keeping careful records, you can gather important information about the nature of the bug that you're chasing. You might even want to use a video capture tool that creates a full-motion video of your interaction with the program.

After you have gathered your evidence, you still should be reluctant to reset the system or the software. Try to reproduce the bug without changing anything that you don't have to. There might be some sort of state that's critical for reproducing the bug. If you can reproduce it now but not later, you know that there is some sort of state involved, but at least you'll know that you saw the bug more than once. Save any input data that you're using, in case you make changes later that cause the bug to mysteriously go away.

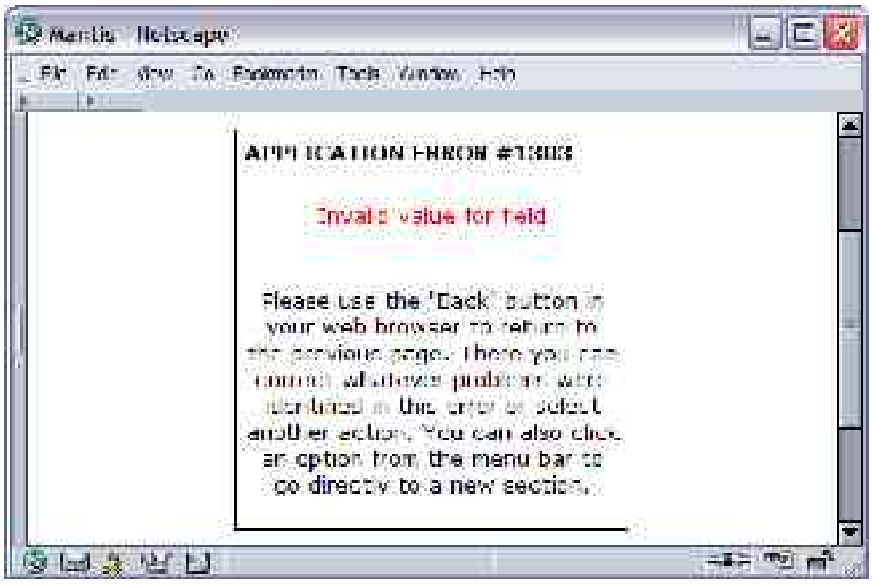
What if you simply can't reproduce the problem after the first occurrence? You should report the bug anyway, using all the information you know. Unless your organization persecutes people who report unreproducible bugs, you want to make a record of what happened. Clearly indicate in the report that you were not able to reproduce the bug. Someone may be able to gather further information if the bug recurs later.

With several years of testing experience, you can develop a sense for which bugs will be difficult to reproduce. For example, when I get a crash that comes out of the blue, perhaps while not giving the application any input at all or just typing text, I've learned that I'm not likely to reproduce the bug easily. You'll notice other patterns as you get experience with bug isolation.

If you've reproduced the bug several times, and you're getting the same failure each time, you can start to play with the state. Here's the ideal – completely reinstall the software under test and the operating system on your computer and all servers involved, restart everything, and then show that you can still reproduce the bug. Now realistically, the times you'll need to go that far will be rare if ever. But remember that if you don't go that that extreme, there might be something about the current state of the system that's necessary for reproducing the bug but isn't there when the programmer tries to find the bug later. So consider restarting the application, restarting the software on the server, and perhaps even rebooting one or more of the systems involved. Or, if you have access to a different system running the software, try to reproduce the bug there. The more varied the scenarios you've seen the bug, and the less your current state is affected by the states the system happened to be in when you were testing, the more likely it is that you have a simple bug that can be easily reproduced.

I'll use an example of a real bug I recently found in the Mantis bug tracking system as an example through the next several sections. Hopefully it won't be too confusing talking about reporting bugs that are in the bug tracking system itself. I was testing Mantis during the course of writing a review. Mantis has a web-based user interface.

> *I had been doing exploratory testing on Mantis for a while. When I try to submit a new bug, I get this error:*



APPLICATION ERROR #1303

Invalid value for field

Please use the 'Back' button in your web browser to return to the previous page. There you can correct whatever problems were identified in this error or select another action. You can also click an option from the menu bar to go directly to a new section.

> *I copy the text of the error and paste it into a text editor that I'm using to take notes. I click the Back button and try to submit the bug again, and I get the exact same error. I do it one more time, and still the same error. So it looks like I have a reproducible bug, at least with the current state of the system.*

## Simplify

Think about the sequence of events that you just took the software through to eventually get to the problem. Ideally, you started testing by clicking one button, and then saw the problem immediately. More likely, though, you had been testing for a while, possibly for hours. Perhaps the last thing you did is the only thing required to reproduce the bug, or maybe you have to repeat hours of testing. Until you can prove that a simpler scenario is sufficient, you have to assume that every detail of your testing session is relevant. Your task is to rule out as many of those details as you can as not being relevant to the problem.

There are many approaches that you can take to simplify your scenario. You want to form hypotheses and then conduct experiments to prove or disprove those hypotheses. Here's a suggested approach that I've refined over the course of reporting hundreds of bugs.

### What to simplify?

There are several different things that are subject to simplification. Consider:

- Procedures. This is usually what testers focus on – shortening the step-by-step interaction with the system.
- Inputs. This is all the data that you feed to the program, such as a command-line argument, a text field in a GUI interface, a file, or database. You want to also reduce this data to the smallest data set that still reproduces the problem.
- Configuration. What options have you selected that are different from the default configuration? If you can't

reproduce the problem the way the software is configured out of the box, find the few crucial settings that are necessary for the problem to show up.

- Platforms. Can you reproduce the problem on all of the operating systems, operating system versions, and hardware combinations that are supported? If not, then you've found an important clue. Also, what about other software that is running, and their versions? Many bugs are not platform-specific, and testing on other platforms can sometimes be difficult, so this area often isn't thoroughly explored.
- Other state information. The items above probably don't capture every possible relevant variable. Look for other things that might vary from one system to another and cause the bug to manifest on some systems but not others.

*In our example, there were the procedures I followed to submit the bug, plus many things I had done before that that might be relevant. There were inputs I typed into the web form, plus the existing data in the database that might be affecting the results. There were a large number of configuration variables that I could set on the server, though I had only modified a few of them. Also, I had added a custom field using the administration interface.*

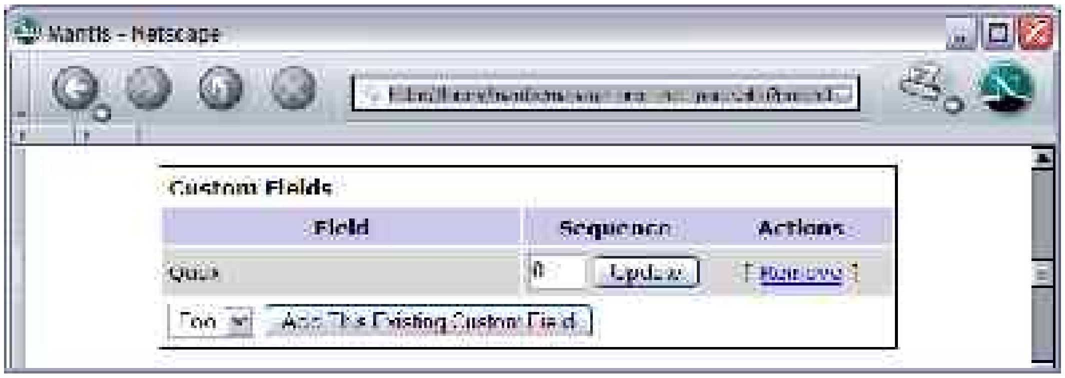
*Mantis can run on many different operating systems and hardware configurations, and it requires a web server (several different ones will do), MySQL database, and PHP, each of which has several supported versions. I was using a 100 megabit network with only a router in between; no firewall or switches involved. I had tried unsuccessfully to install Mantis on a different system, so I didn't have another test bed easily accessible. I did have access to a production Mantis installation—the one used to track bugs against Mantis itself, but it wouldn't have been appropriate to corrupt that database with test data.*

## Are you feeling lucky?

You might have a hunch about a particular small part of your scenario that is the key for reproducing the problem. If so, go ahead and try those steps in isolation, and you might be able to remove a large chunk of irrelevant details very quickly. This is what we referred to as the “bottom-up” approach in the “Software Defect Isolation” paper.

These hunches often lead to dead ends, though, so if it turns out that your simplified scenario does not reproduce the bug, you've probably removed a necessary step somewhere. The best thing to do at this point is to take a more systematic approach. Otherwise, you could spend a lot of time following an unproductive path.

*In our example, I remember that I had recently created a custom field, and I 'm pretty sure that this is the first time I had tried to report a bug since then. So I have a hunch that this custom field is an important factor in what's going on. I remove the custom field from the project, and sure enough, I can submit a bug without seeing the error. So I'm well on my way toward figuring this out.*



Custom Fields

| Field | Sequence | Actions |
|---|---|---|
| Quick | 0 Update | [ Remove ] |

Foo — Add This Existing Custom Field

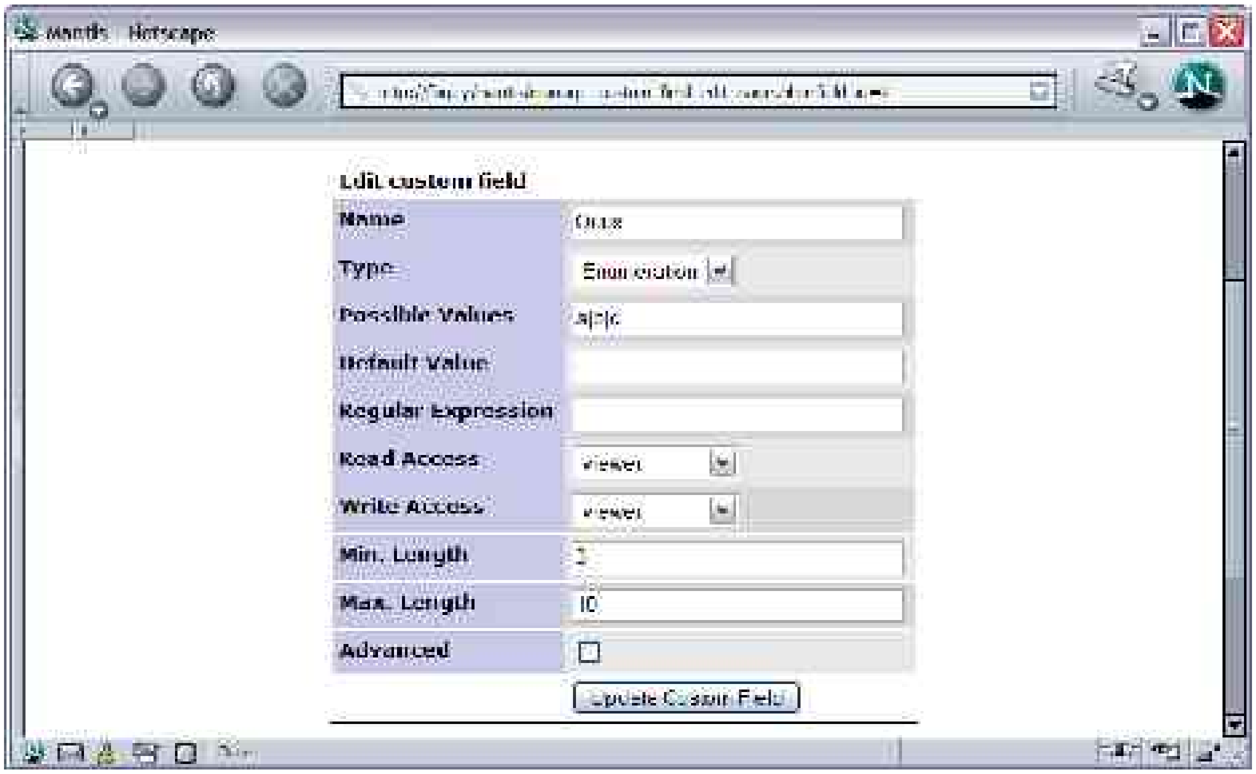
## The top-down approach

When your hunches haven't reduced your scenario sufficiently, it's time to take a more objective approach. The top-down approach is a systematic method for removing irrelevant details from your scenario, and it takes a lot of discipline to do well. I even try to think like a computer when I do this, to guard against getting trapped by assumptions I might make that would send me on a a wild goose chase.

Consider all of the factors involved as you simplify—procedures, inputs, configuration, and platform. Don't change too many variables at once, or you could get confused. You want track two scenarios—the simplest scenario so far that reproduces the bug, and the most complex scenario that doesn't reproduce the bug. Your goal then is to get these two scenarios to converge so that the only difference between them is the critical variables that you need to know to reproduce the bug.

If a feature is completely broken, then you won't find a scenario involving that feature that doesn't reproduce the bug. It's important to try several different ways to get the feature to work before you declare it completely inoperable.

*In our example, we have the scenario without the custom field where we don't see the bug, and the one with the custom field where we're hitting the bug. I turn the custom field back on, and confirm that I get the same error. So I go look at the details of how I set up the custom field to see if I get can a clue about what's going on:*



Edit custom field

| | |
|---|---|
| Name | Quux |
| Type | Enumeration |
| Possible Values | a\|b\|c |
| Default Value | |
| Regular Expression | |
| Read Access | viewer |
| Write Access | viewer |
| Min. Length | 2 |
| Max. Length | 10 |
| Advanced | ☐ |

*Hmmm, any clues here? Say, what does "Min. Length" and "Max. Length" mean here? I change the Min. Length to 1, try to submit a bug, and sure enough, I don't get the error. I change it to 2 and confirm that I get the error again. Now I've found the threshold, and it's pretty clear that the Min. Length is being applied to the size of the enumeration value ("a", "b", or "c"). I then remove the value for Max. Length and reproduce the bug again, which further simplifies the scenario.*

*I could also put just one or two values in the enumeration - "a" or "a|b" instead of "a|b|c", but that seems to be overkill, and just having "a" makes it look like a scenario that a real user wouldn't try.*

*I report bug 3793 against Mantis (http://bugs.mantisbt.org/bug_view_page.php?bug_id=0003793):*

```
Summary: Enumeration values might be impossible to select

Description: On the Manage->Manage Custom Fields->Edit Custom Field
page, it's possible to create a value for an enumeration that can never
be selected. This happens if the length of the enumeration value is
less than the Min. Length, longer than the Max. Length, or doesn't
```

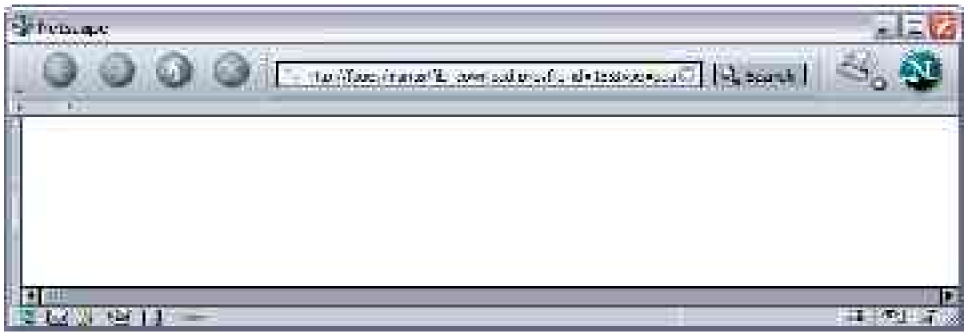
```
match the regular expression.

These checks don't make sense for enumerations. Consider issuing a
warning if they are defined when an enumeration is selected as the data
type for the custom field.

The result when selecting an enumeration value that fails these checks
is the unhelpful message:

APPLICATION ERROR 0001303
Invalid value for field
```

## The bisection technique

Bisection, also called a binary search, is an efficient algorithm for searching sorted lists. But with some creative thinking, it can also be a big help with bug isolation. Here's how you'd use the algorithm to find a word in the dictionary: open the dictionary to the page that's in the middle of the book. Check to see if the word is earlier or later in the dictionary. Then find the middle of the first half or last half of the book, depending on which direction you need to go. Repeat, by checking the middle of successively smaller and smaller sections, until you get to the right page. You'll find your word quickly, especially compared to searching linearly starting at page 1.

Now let's apply it to testing. Let's say you generate a data file full of a million random characters, and your application crashes when you give that file as input. You should try to find the smallest possible file that reproduces the crash. After all, a bug report about a 10 or 1 byte file causing a crash is much more impressive than a bug report about the large file. Using the bisection technique, you can quickly narrow down the exact file size that reproduces the bug, such that a file that's one byte smaller works fine.

*As another real example, consider this testing session for Mantis that illustrates one of my favorite attacks. I use a Perl script one-liner to generate a text file containing a million "x" characters on a single line. Then I upload that file as an attachment to a Mantis bug. It uploads okay, but when I try to download the attachment, the browser just gives me a blank screen. (Good thing I didn't assume all was fine when the upload worked.) This is what it looks like—notice the horizontal scroll bar:*



*Here's what I did to find the smallest file size that fails, false starts and all. Each item lists the file size I used and the results. Each iteration takes less than half a minute.*

*1,000,000–Empty page. I notice that Mantis displays the size of the attachments, which gives a nice verification that I got the file size right. While the bisection technique says I should try 500,000 next, instead I follow my intuition that the number might be orders of magnitude smaller than a million.*

*10,000–Empty page. I copy file path into the clipboard so I can easily enter it into the upload form next time. I notice that I inadvertently verified that Mantis can load multiple attachments with the same file name. Next I'll make another big jump.*

*100–Works. All the characters are visible. I'll indicate the largest working file size (100) and the smallest problem file size (10,000) as "[100-10,000]" as we narrow it down.*

*1,000–Works. The range is now [1,000-10,000].*

*5,000–Empty page. [1,000-5,000] I click and drag on the area of the browser window where the text would be, and notice that it actually shows up when it's highlighted. So we're probably looking at a browser bug here. I load the data file I generated directly from the local disk into the browser, and it displays fine. Interesting. But I want to get this number narrowed down before I explore that any further.*

*3,000–Works. [3,000-5,000]*

*4,000–Works. [4,000-5,000]*

*4,500–The text is again invisible. [4,000-4,500]*

*4,200–Invisible. [4,000-4,200]*

*4,100–Invisible. [4,000-4,100]*

*4,096–Invisible. [4,000-4,096] I realize that we're close to a power of two, which is a likely place for a bug to lurk. 4096 is $2^{12}$.*

*4,095–Invisible [4,000-4,095] So the bug isn't right on the power of two boundary. Back to a more mechanical approach.*

*4,050–Works. [4,050-4,095] I still think it could be close to 4,096, so I'll jump further in that direction.*

*4,080–Works. [4,080-4,095] We're getting close now.*

*4,090–Works. [4,090-4,095]*

*4,093–Works. [4,093-4,095]*

*4,094–Works. [4,094-4,095] Looks like we've found the critical point, one off from a power of two boundary. Let's check the other side of the boundary to verify that it isn't moving.*

*4,095–Invisible. Yup, we got it! We can report that a line 4,095 characters long or longer doesn't render properly in the browser. But not yet, so stay tuned.*

For a stranger case of the results of the bisection method, see bug 3798 that I filed against Mantis, "Locked out of View Bugs page after large query" at http://bugs.mantisbt.org/bug_view_page.php?bug_id=0003798. I narrowed the test case down to a precise failure point, but after restarting the web server, the failure point moved. I suspect that cases like this occur when the failure depends on how many system resources are available, such as memory. The only way I could have found this out was by determining the precise failure point each time, and the bisection method made this fairly easy to do.

By the way, there are opportunities for automating a test using the bisection technique to speed up the isolation process even further. Once when I was teaching a tutorial, one of the participants suggested that we use automation. While I didn't want to code the finer points of the bisection algorithm on the spot, we did come up with a simple linear search script within only a few minutes that was able to get very close to the critical point that we needed to find.

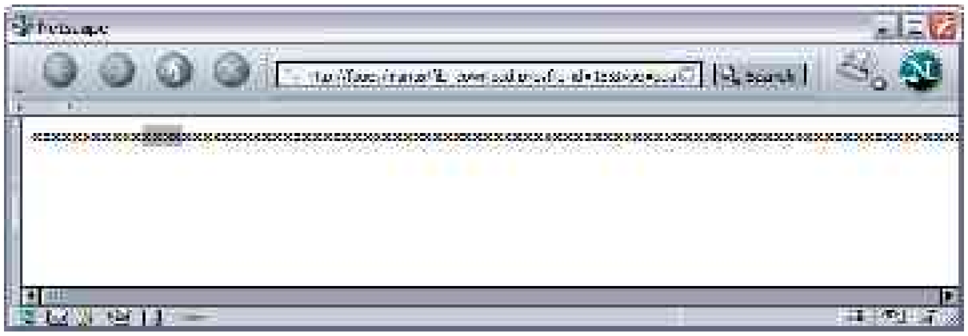
## Refinement

Once we have isolated the bug, we still have the opportunity to discover additional information about it that will help with characterizing the severity of the bug and jump start the debugging process.

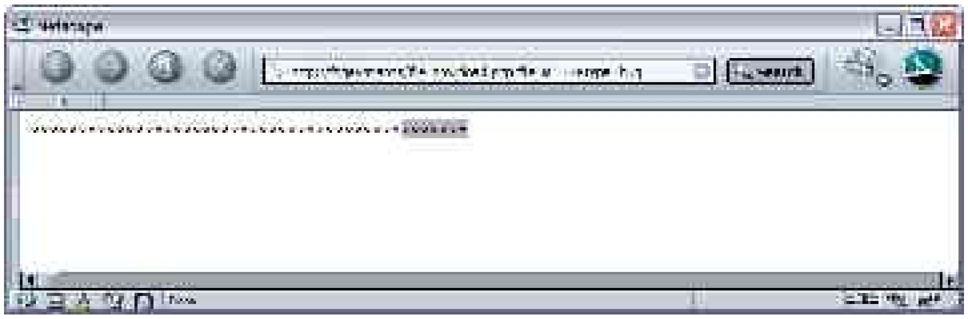
Let's go right back to the example where we left off. Again, the numbers on the left indicate the size of the file that I upload.

*When I check the 4,095 character file again, I notice that when I highlight a character, all of the characters on the page become visible. Earlier I remember that only the characters to the left of the selected text was visible, and I don't remember when the behavior changed. So I decide to go back and track this down, at first using the attachments I've already created.*

*4,096–All text is visible when a few characters are selected. It looks like this:*



*5,000–Text is only visible to the left of the selection. I see this:*



*4,500–Only visible to the left.*

*4,200–Only visible to the left.*

*4,100–All text is visible when I make a small selection. I need to upload additional files from here on because I'm converging on a different range than before.*

*4,150–All text is visible when I make a small selection.*

*4,175–Only visible to the left.*

*4,160–All text is visible when I make a small selection.*

*4,170–Only visible to the left.*

*4,165–Only visible to the left.*

*4,168–Only visible to the left.*

*4,169–Only visible to the left. I'm back to 4,170, which I already tested. Wait a minute! 4,170 gets the same result as 4,169. I've somehow gone in the wrong direction. I got sloppy by not keeping careful records. The range is [4,160-4,165].*

*4,162–Only visible to the left. [4,160-4,162]*

*4,161–Only visible to the left. [4,160-4,161] It looks like we have it! Let's make sure.*

*4,160–Only visible to the left. No fair! That's not the same result I got last time. Sometimes a target will move while I'm testing, and sometimes my record keeping is just wrong. So I'll back up and see if anything else is different now.*

*4,150–All text is visible when I make a small selection. Same as before.*

*4,155–Only visible to the left. Looks like the range is [4,150-4,155] now.*

*4,152–Only visible to the left. [4,150-4,152]*

*4,151–Only visible to the left. [4,150-4,151] Do we have it now? I don't get my hopes up.*

*4,150–All text is visible when I make a small selection. All looks okay, but I'm still suspicious.*

*4,151–All text is visible when I make a small selection – a different result than before! I close several extraneous browser windows to see if this might be a memory-related problem.*

*4,151–Aha! I discover that the result depends on how I select. If I drag the selection right far enough, all text becomes visible. I was being sloppy with how I was selecting text. I should have done it exactly the same way every time.*

*5,000–All text is visible if I select 904 characters or more starting on the left. A quick calculation shows that this leaves exactly 4096 characters to the right, which is right at a power of 2.*

*4,095–I just need to select one character to get all the text to show up.*

*4,096–The text to the right is displayed if I select the two characters at left end, or one character elsewhere.*

*4,100–I can make all the text visible if I select any character from sixth position from the left onward. I think I see the pattern.*

*At this point I copy the data file to the web server and view it as a static file instead of going through Mantis. That reproduces the bug, so I know it has nothing to do with Mantis. It looks like the bug is in my browser, Netscape. I don't hold out much hope of getting a commercial vendor to care about a bug report like this, so I install the latest version of Mozilla and find that it has the same problem. I check the bug database for Mozilla and find several similar, though not identical, issues already filed. I decide that bug 92193, "Very long line is not fully rendered," is close enough and I will add a comment there (http://bugzilla.mozilla.org/show_bug.cgi?id=92193).*

*Getting ready to report my findings almost always inspires me to conjure up a few more experiments. Based on comments about Linux in the bugs I looked at, I tried accessing the test file from a Windows-based web server, proving that the problem isn't specific to a Linux web server. Comments for this bug report point out an interesting environmental variation that I didn't try before–the thresholds move if I change the font size—which I confirm with my test data but don't try to characterize further. I also think to try double-clicking the invisible text, which gives interesting results. And one more thing, in an attempt to see if this is an underlying operating system problem, I try to reproduce the bug with Internet Explorer, and indeed, it also has trouble with disappearing text with very long lines. But the symptoms aren't quite the same, and the line has to be much longer than with Netscape and Mozilla. I decide to make a note of this but not explore further.*

*I see that no one else has isolated the bug to the extent that I have, so I file this comment on Mozilla bug 92193:*

*I just isolated a bug in Mozilla 1.7 (originally seen in Netscape 7.1) that seems to be this same issue. I'm running on Windows XP. I'm viewing text files hosted on an Apache server - I tried Apache on both XP and Linux with the same results.*

*The text files contain a single long line of "x" characters. If they contain 4095 or more characters, the page comes up blank. (I'll attach this one as a test case.) Then if I select any character by blindly clicking where the text should be, they all become visible. When I deselect, all goes blank again.*

*For a line 4096 characters long, if I select any character from the second one onward, the text becomes visible. If I select the first character, only that one is visible. With 4100 characters, when I select any character from the sixth onward, all becomes visible. But if I select any of characters 1-5, only the selected characters and those to the left of them are visible. This pattern continues for long lines.*

> *If I double-click over the invisible text, it's all highlighted as a solid gray rectangle with the text still not visible.*
>
> *The thresholds change as I zoom in and out with "Ctrl +" and "Ctrl -".*
>
> *I've noticed a vaguely similar problem with IE 6 rendering long lines in text files, but the problem occurs at a much higher threshold.*

You can see from the example that there may be many opportunities to refine your bug report. I found the precise breaking point for the software, the smallest (and therefore most realistic) data file that illustrated the problem. Then I found a secondary effect and characterized it through a combination of mechanical application of the bisection technique, intuition, and trial and error.

One thing to watch out for is when there is more than one failure mode at different points in a continuum. Especially when I'm doing robustness testing, it's not uncommon to find a moderate failure at one point, and a more serious failure with more extreme test data. For more on finding the most serious consequence of a bug, see Cem Kaner's "Bug Advocacy" presentation, pages 17-25, and Rex Black's “The Bug Reporting Processes” paper.

## Where to draw the line?

There is no clear dividing line between the bug isolation and reporting that the tester does and the debugging that the programmer does. In most organizations, there's plenty of room for the testers to put more effort into bug isolation. On the other hand, there may also be reasons to have the testers report the first symptom they see and then move on. Here are some factors in favor of having testers put significant effort into isolating the bugs they report:

- Testers may be more motivated than the programmers to show that the bug has a real impact and should be fixed.
- Because they spend more time on bug reporting than programmers, the testers may be better as bug isolation than the programmers.
- The information that the tester learns about the bug can be put into immediate use for further testing.
- Testers may have access to more resources, such as staff and lab equipment, than the programmers.

However, there are also factors that could indicate that it's best to put most of the weight of bug isolation on the programmers:

- The testers may not be highly skilled or experienced, so they may not be adept at bug isolation.
- Testers might put significant effort into isolating a bug that turns out to be a low priority for fixing.
- Testing resources may be more constrained than development resources.
- It might be more productive to use knowledge of the code to zoom in on the problem.
- Having the testers do significant bug isolation makes their somewhat unpredictable project schedule even more fluid, because it's hard to predict how many bugs will be found and how hard they are to characterize. Ofcourse, the programmers have the same problem if we shift the work to them instead.

> *Looking back at the last example, you can see a conscientious effort to isolate the bug that went beyond reporting the first observed symptom. In this case, it looks like a good choice. The bug report that was amended had been open for almost three years, and the information already in the report did not characterize the problem well. The new information may lead to some long-awaited progress on this bug.*
>
> *There were still some possibilities for exploration that were stopped short. There are always judgment calls like this, because the opportunities for further exploration are as endless as the number of possible test cases. In this case, there was a correlation with the font size that I acknowledged, but I didn't characterize its effects on the thresholds. This might have led to a theory about the total width of the rendered page being a factor, but I chose to hand off that line of reasoning to the programmer. Also, the fact that Internet Explorer has a similar bug was surprising, and could possibly indicate that a flaw in the operating system is at fault. I decided that the programmer could better judge this because she would know more about the operating system interfaces that were in use. I didn't check other platforms, mostly because the bug was not*

*severe and I felt my time was better spent elsewhere.*

In most of the organizations I've observed, I believe the testers should have been doing more bug isolation, especially for severe bugs. I recommend that you put conscious thought into the factors listed above and help your organization decide on guidelines for where the draw the line.

## References

[Black 2000a] “The Bug Reporting Processes,” Rex Black, *Journal of Software Testing Professionals*, 2000, http://rexblackconsulting.com/publications/Bug%20Reporting%20Processes%20(Article).pdf

[Black 2000b] “The Fine Art of Writing a Good Bug Report,” Rex Black, Quality Week 2000, paper – http://rexblackconsulting.com/publications/Fine%20Art%20of%20Writing%20a%20Good%20Bug%20Report%20(Paper).pdf, slides – http://rexblackconsulting.com/publications/Fine%20Art%20of%20Writing%20a%20Good%20Bug%20Report%20(Slides).pdf

Cem Kaner, "Bug Advocacy," July, 2000, http://kaner.com/pdfs/bugadvoc.pdf

Cem Kaner, (2002) "Effective bug reporting." (Tutorial session) *15th International Software Quality Conference (Quality Week),* San Francisco, CA, September, 2002, http://kaner.com/pdfs/BugAdvocacy.pdf

Faught, Danny R. and Prathibha Tammana. “Software Defect Isolation.” High-Performance Computing Users Group, March 1998, InterWorks, April 1998. http://tejasconsulting.com/papers/iworks98/defect_isol.pdf

Faught, Danny R., “A Bug Tracking Story.” ASEE Software Engineering Process Improvement Workshop, 2002. http://tejasconsulting.com/papers/bug_tracking_story.pdf

Faught, Danny R., “Bug Report: But It's a Feature!” *STQE Magazine,* March/April 2003. http://tejasconsulting.com/stqe/itsafeature.pdf

Faught, Danny R., “A Lesson in Scripting,” *STQE* Magazine, March/April 2002. http://tejasconsulting.com/stqe/a_lesson_in_scripting.pdf (re: Perl one-line test data generator)