Producing Open Source Software

How to Run a Successful Free Software Project

Karl Fogel

Chapter 6. Communications

An open source project must do many things: recruit users and developers, encourage new contributors to become more deeply involved, allow free-flowing discussion while still reaching necessary decisions, maintain a body of knowledge and convention that guides newcomers and experts alike, and, of course, produce working software.

Coordinate people to accomplish all this together requires many techniques, and because open source collaboration is ultimately based on software code, most of those techniques revolve around the written word. We'll start there.

Written Culture

The ability to write clearly is perhaps the most important skill one can have in an open source environment. In the long run it matters more than programming talent. A great programmer with lousy communications skills can get only one thing done at a time, and even then may have trouble convincing others to pay attention. But a lousy programmer with good communications skills can coordinate and persuade many people to do many different things, and thereby have a significant effect on a project's direction and momentum.

There does not seem to be much correlation, in either direction, between the ability to write good code and the ability to communicate with one's fellow human beings. There is some correlation between programming well and describing technical issues well, but describing technical issues is only a tiny part of the communications in a project. Much more important is the ability to empathize with one's audience, to see one's own posts and comments as others see them, and to cause others to see their own posts with similar objectivity. Equally important is noticing when a given medium or communications method is no longer working well, perhaps because it doesn't scale as the number of users increases, and taking the time to do something about it.

All of which is obvious in theory — what makes it hard in practice is that free software development environments are bewilderingly diverse both in audiences and in communications mechanisms. Should a given thought be expressed in a post to the mailing list, as an annotation in the bug tracker, or as a comment in the code? When answering a question in a public forum, how much knowledge can you assume on the part of the reader, given that "the reader" is not only the one who asked the question in the first place, but all those who might see your response? How can the developers stay in constructive contact with the users, without getting swamped by feature requests, spurious bug reports, and general chatter? How do you tell when a medium has reached the limits of its capacity, and what do you do about it?

Solutions to these problems are usually partial, because any particular solution is eventually made obsolete by project growth or changes in project structure. They are also often *ad hoc*, because they're improvised responses to dynamic situations. All participants need to be aware of when and how communications can become bogged down, and be involved in solutions. Helping people do this is a big part of managing an open source project.

The sections that follow discuss both how to conduct your own communications, and how to make maintenance of communications mechanisms a priority for everyone in the project.[1]

[1]There has been some interesting academic research on this topic; for example, see *Group Awareness in Distributed Software Development* by Gutwin, Penner, and Schneider. This paper was online for a while, then unavailable, then online again at http://www.st.cs.uni-sb.de/edu/empirical-se/2006/PDFs/gutwin04.pdf. So try there first, but be prepared to use a search engine if it moves again.

You Are What You Write

Consider this: the only thing anyone knows about you on the Internet comes from what you write, or what others write about you. You may be brilliant, perceptive, and charismatic in person — but if your emails are rambling and unstructured, people will assume that's the real you. Or perhaps you really are rambling and unstructured in person, but no one need ever know it, if your posts are lucid and informative.

Devoting some care to your writing will pay off hugely. Long-time free software hacker Jim Blandy tells the following story:

> Back in 1993, I was working for the Free Software Foundation, and we were beta-testing version 19 of GNU Emacs. We'd make a beta release every week or so, and people would try it out and send us bug reports. There was this one guy whom none of us had met in person but who did great work: his bug reports were always clear and led us straight to the problem, and when he provided a fix himself, it was almost always right. He was top-notch.
>
> Now, before the FSF can use code written by someone else, we have them do some legal paperwork to assign their copyright interest to that code to the FSF. Just taking code from complete strangers and dropping it in is a recipe for legal disaster.
>
> So I emailed the guy the forms, saying, "Here's some paperwork we need, here's what it means, you sign this one, have your employer sign that one, and then we can start putting in your fixes. Thanks very much."
>
> He sent me back a message saying, "I don't have an employer."
>
> So I said, "Okay, that's fine, just have your university sign it and send it back."
>
> After a bit, he wrote me back again, and said, "Well, actually... I'm thirteen years old and I live with my parents."

Because that kid didn't write like a thirteen-year-old, no one knew that's what he was. Following are some ways to make your writing give a good impression too.

Structure and Formatting

Don't fall into the trap of writing everything as though it were a cell phone text message. Write in complete sentences, capitalizing the first word of each sentence, and use paragraph breaks where needed. This is most important in emails and other composed writings. In IRC or similarly ephemeral forums, it's generally okay to leave out capitalization, use compressed forms of common expressions, etc. Just don't carry those habits over into more formal, persistent forums. Emails, documentation, bug reports, and other pieces of writing that are intended to have a permanent life should be written using standard grammar and spelling, and have a coherent narrative structure. This is not because there's anything inherently good about following arbitrary rules, but rather that these rules are *not* arbitrary: they evolved into their present forms because they make text more readable, and you should adhere to them for that reason. Readability is desirable not only because it means more people will understand what you write, but because it makes you look like the sort of person who takes the time to communicate clearly: that is, someone worth paying attention to.

Careful grammar also minimizes ambiguity. This is especially important in technical writing, where plausible alternatives will often be juxtaposed, and the distinction between cause and effect may not be

immediately clear from context alone. A grammatical structure that represents these things in precisely the way the writer intended will help everyone avoid confusion.

For email in particular, experienced open source developers have settled on certain formatting conventions:

- Send plain text mails only, not HTML, RichText, or other formats that might get mangled by certain online archives or text-based mail readers. When including screen output, snippets of code, or other preformatted text, offset it clearly, so that even a lazy eye can easily see the boundaries between your prose and the material you're quoting. If the overall structure of your post is still visible from five meters away, you're doing it right.

- For preformatted blocks, such as quoted code or error messages, try to stay under 80 columns wide, which has become the *de facto* standard terminal width (that is, some people may use wider displays, but no one uses a narrower one). By making your lines a little *less* than 80 columns, you leave room for a few levels of quoting characters to be added in others' replies without forcing a rewrapping of your preformatted text.

- When quoting someone else's mail, insert your responses where they're most appropriate, at several different places if necessary, and trim off the parts of their mail you didn't use. If you're writing a quick response that applies to their entire post, and your response will be sensible even to someone who hasn't read the original, then it's okay to *top-post* (that is, to put your response above the quoted text of their mail); otherwise, quote the relevant portion of the original text first, followed by your response.

- *Construct the subject lines of new mails carefully*. It's the most important line in your mail, because it allows each other person in the project to decide whether or not to read more. Modern mail reading software organizes groups of related messages into threads, which can be defined not only by a common subject, but by various other headers (which are sometimes not displayed). It follows that if a thread starts to drift to a new topic, you can — and should — adjust the subject line accordingly when replying. The thread's integrity will be preserved, due to those other headers, but the new subject will help people looking at an overview of the thread know that the topic has drifted. Likewise, if you really want to start a new topic, do it by posting a fresh mail, not by replying to an existing mail and changing the subject. Otherwise, your mail would still be grouped in to the same thread as what you're replying to, and thus fool people into thinking it's about something it's not. Again, the penalty would not only be the waste of their time, but the slight dent in your credibility as someone fluent in using communications tools.

Content

Well-formatted mails attract readers, but content keeps them. No set of fixed rules can guarantee good content, of course, but there are some principles that make it more likely.

Make things easy for your readers. There's a ton of information floating around in any active open source project, and readers cannot be expected to be familiar with most of it — indeed, they cannot always be expected to know how to become familiar. Wherever possible, your posts should provide information in the form most convenient for readers. If you have to spend an extra two minutes to dig up the URL to a particular thread in the mailing list archives, in order to save your readers the trouble of doing so, it's worth it. If you have to spend an extra 5 or 10 minutes summarizing the conclusions so far of a complex thread, in order to give people context in which to understand your post, then do so. Think of it this way: the more successful a project, the higher the reader-to-writer ratio in any given forum. If every post you make is seen by *n* people, then as *n* rises, the worthwhileness of expending extra effort to save those people time rises with it. And as people see you imposing this standard on yourself, they will work to match it in their own communications. The result is, ideally, an increase in the global efficiency

of the project: when there is a choice between *n* people making an effort and one person doing so, the project prefers the latter.

Don't engage in hyperbole. Exaggerating in online posts is a classic arms race. For example, a person reporting a bug may worry that the developers will not pay sufficient attention, so he'll describe it as a severe, showstopper problem that is preventing him (and all his friends/coworkers/cousins) from using the software productively, when it's actually only a mild annoyance. But exaggeration is not limited to users — programmers often do the same thing during technical debates, particularly when the disagreement is over a matter of taste rather than correctness:

> "Doing it that way would make the code totally unreadable. It'd be a maintenance nightmare, compared to J. Random's proposal..."

The same sentiment actually becomes *stronger* when phrased less sharply:

> "That works, but it's less than ideal in terms of readability and maintainability, I think. J. Random's proposal avoids those problems because it..."

You will not be able to rid the project of hyperbole completely, and in general it's not necessary to do so. Compared to other forms of miscommunication, hyperbole is not globally damaging — it hurts mainly the perpetrator. The recipients can compensate, it's just that the sender loses a little more credibility each time. Therefore, for the sake of your own influence in the project, try to err on the side of moderation. That way, when you *do* need to make a strong point, people will take you seriously.

Edit twice. For any message longer than a medium-sized paragraph, reread it from top to bottom before sending it but after you think it's done the first time. This is familiar advice to anyone who's taken a composition class, but it's especially important in online discussion. Because the process of online composition tends to be highly discontinuous (in the course of writing a message, you may need to go back and check other mails, visit certain web pages, run a command to capture its debugging output, etc.), it's especially easy to lose your sense of narrative place. Messages that were composed discontinuously and not checked before being sent are often recognizable as such, much to the chagrin (or so one would hope) of their authors. Take the time to review what you send. The more your posts hold together structurally, the more they will be read by others.

Tone

After writing thousands of messages, you will probably find your style tending toward the terse. This seems to be the norm in most technical forums, and there's nothing wrong with it per se. A degree of terseness that would be unacceptable in normal social interactions is simply the default for free software hackers. Here's a response I once drew on a mailing list about some free content management software, quoted in full:

```
Can you possibly elaborate a bit more on exactly what problems
you ran into, etc?

Also:

What version of Slash are you using? I couldn't tell from your
original message.

Exactly how did you build the apache/mod_perl source?

Did you try the Apache 2.0 patch that was posted about on
slashcode.com?
```

```
Shane
```

Now *that's* terse! No greeting, no sign-off other than his name, and the message itself is just a series of questions phrased as compactly as possible. His one declarative sentence was an implicit criticism of my original message. And yet, I was happy to see Shane's mail, and didn't take his terseness as a sign of anything other than him being a busy person. The mere fact that he was asking questions, instead of ignoring my post, meant that he was willing to spend some time on my problem.

Will all readers react positively to this style? Not necessarily; it depends on the person and the context. For example, if someone has just posted acknowledging that he made a mistake (perhaps he wrote a bug), and you know from past experience that this person tends to be a bit insecure, then while you may still write a compact response, you should make sure to leaven it with some sort of acknowledgement of his feelings. The bulk of your response might be a brief, engineer's-eye analysis of the situation, as terse as you want. But at the end, sign off with something indicating that your terseness is not to be taken as coldness. For example, if you've just given reams of advice about exactly how the person should fix the bug, then sign off with "Good luck, <your name here>" to indicate that you wish him well and are not mad. A strategically placed smiley face or other emoticlue can often be enough to reassure an interlocutor, too.

It may seem odd to focus as much on the participant's feelings as on the surface of what they say, but, to put it baldly, feelings affect productivity. Feelings are important for other reasons too, but even confining ourselves to purely utilitarian grounds, we may note that unhappy people write worse software and tackle fewer bugs. Given the restricted nature of most electronic media, though, there will often be no overt clue about how a person is feeling. You will have to make an educated guess based on a) how most people would feel in that situation, and b) what you know of this particular person from past interactions. Some people prefer a more hands-off attitude, and simply deal with everyone at face value, the idea being that if a participant doesn't say outright that he feels a particular way, then one has no business treating him as though he does. I don't buy this approach, for a couple of reasons. One, people don't behave that way in real life, so why would they online? Two, since most interactions take place in public forums, people tend to be even more restrained in expressing emotions than they might be in private. To be more precise, they are often willing to express emotions directed at others, such as gratitude or outrage, but not emotions directed inwardly, such as insecurity or pride. Yet most humans work better when they know that others are aware of their state of mind. By paying attention to small clues, you can usually guess right most of the time, and motivate people to stay involved to a greater degree than they otherwise might.

I don't mean, of course, that your role is to be a group therapist, constantly helping everyone to get in touch with their feelings. But by paying careful attention to long-term patterns in people's behavior, you will begin to get a sense of them as individuals even if you never meet them face-to-face. And by being sensitive to the tone of your own writing, you can have a surprising amount of influence over how others feel, to the ultimate benefit of the project.

Recognizing Rudeness

One of the defining characteristics of open source culture is its distinctive notions of what does and does not constitute rudeness. While the conventions described below are not unique to free software development, nor even to software in general — they would be familiar to anyone working in mathematics, the hard sciences, or engineering disciplines — free software, with its porous boundaries and constant influx of newcomers, is an environment where these conventions are especially likely to be encountered by people unfamiliar with them. (This is one reason why it's good to be generous when trying to figure out whether someone has violated the code of conduct, in a project that has one — see the section called “Codes of Conduct” in Chapter 2, *Getting Started*.)

Let's start with the things that are *not* rude:

Technical criticism, even when direct and unpadded, is not rude. Indeed, it can be a form of flattery: the critic is saying, by implication, that the target is worth taking seriously, and is worth spending some time on. That is, the more viable it would have been to simply ignore someone's post, the more of a compliment it becomes to take the time to criticize it (unless the critique descends into an *ad hominem* attack or some other form of obvious rudeness, of course).

Blunt, unadorned questions, such as Shane's questions to me in the previously quoted email, are not rude either. Questions that in other contexts might seem cold, rhetorical, or even mocking, are often intended seriously, and have no hidden agenda other than eliciting information as quickly as possible. The famous technical support question "Is your computer plugged in?" is a classic example of this. The support person really does need to know if your computer is plugged in, and after the first few days on the job, has gotten tired of prefixing her question with polite blandishments ("I beg your pardon, I just want to ask a few simple questions to rule out some possibilities. Some of these might seem pretty basic, but bear with me..."). At this point, she doesn't bother with the padding anymore, she just asks straight out: is it plugged in or not? Equivalent questions are asked all the time on free software mailing lists. The intent is not to insult the recipient, but to quickly rule out the most obvious (and perhaps most common) explanations. Recipients who understand this and react accordingly win points for taking a broad-minded view without prompting. But recipients who react badly must not be reprimanded, either. It's just a collision of cultures, not anyone's fault. Explain amiably that your question (or criticism) had no hidden meanings; it was just meant to get (or transmit) information as efficiently as possible, nothing more.

So what *is* rude?

By the same principle under which detailed technical criticism is a form of flattery, failure to provide quality criticism can be a kind of insult. I don't mean simply ignoring someone's work, be it a proposal, code change, new ticket filing, or whatever. Unless you explicitly promised a detailed reaction in advance, it's usually okay to simply not react at all. People will assume you just didn't have time to say anything. But if you *do* react, don't skimp: take the time to really analyze things, provide concrete examples where appropriate, dig around in the archives to find related posts from the past, etc. Or if you don't have time to put in that kind of effort, but still need to write some sort of brief response, then state the shortcoming openly in your message ("I think there's a ticket filed for this, but unfortunately didn't have time to search for it, sorry"). The main thing is to recognize the existence of the cultural norm, either by fulfilling it or by openly acknowledging that one has fallen short this time. Either way, the norm is strengthened. But failing to meet that norm, while at the same time not explaining *why* you failed to meet it, is like saying the topic (and those participating in it) was not worth much of your time — that your time is more valuable than theirs. Better to show that your time is valuable by being terse than by being lazy.

There are many other forms of rudeness, of course, but most of them are not specific to free software development, and common sense is a good enough guide to avoid them. See also the section called “Nip Rudeness in the Bud” in Chapter 2, *Getting Started*, if you haven't already.

Face

There is a region in the human brain devoted specifically to recognizing faces. It is known informally as the "fusiform face area" and apparently its capabilities are at least partly inborn, not learned. It turns out that recognizing individual people is such a crucial survival skill that we have evolved specialized hardware to do it.

Internet-based collaboration is therefore psychologically odd, because it involves tight cooperation between human beings who almost never get to identify each other by the most natural, intuitive methods: facial recognition first of all, but also sound of voice, posture, etc. To compensate for this, try to use a consistent *screen name* everywhere. It should be the front part of your email address (the part before the @-sign), your IRC username, your repository committer name, your ticket tracker username,

and so on. This name is your online "face": a short identifying string that serves some of the same purpose as your real face, although it does not, unfortunately, stimulate the same built-in hardware in the brain.

The screen name should be some intuitive permutation of your real name (mine, for example, is "kfogel"). In some situations it will be accompanied by your full name anyway, for example in mail headers:

```
From: "Karl Fogel" <kfogel@whateverdomain.com>
```

Actually, there are two things going on in that example. As mentioned earlier, the screen name matches the real name in an intuitive way. But also, the real name is *real*. That is, it's not some made-up appellation like:

```
From: "Wonder Hacker" <wonderhacker@whateverdomain.com>
```

There's a famous cartoon by Paul Steiner, from the July 5, 1993 issue of *The New Yorker*, that shows one dog logged into a computer terminal, looking down and telling another conspiratorially: "On the Internet, nobody knows you're a dog." This kind of thought probably lies behind a lot of the self-aggrandizing, meant-to-be-hip online identities people give themselves — as if calling oneself "Wonder Hacker" will actually cause people to believe one *is* a wonderful hacker. But the fact remains: even if no one knows you're a dog, you're still a dog. A fantastical online identity never impresses readers. Instead, it makes them think you're more into image than substance, or that you're simply insecure. Use your real name for all interactions, or if for some reason you require anonymity, then make up a name that sounds like a perfectly normal real name, and use it consistently.

In addition to keeping your online face consistent, there are some things you can do to make it more attractive. If you have an official title (e.g., "doctor", "professor", "director"), don't flaunt it, nor even mention it except when it's directly relevant to the conversation. Hackerdom in general, and free software culture in particular, tends to view title displays as exclusionary and a sign of insecurity. It's okay if your title appears as part of a standard signature block at the end of every mail you send, just don't ever use it as a tool to bolster your position in a discussion — the attempt is guaranteed to backfire. You want folks to respect the person, not the title.

Speaking of signature blocks: keep them small and tasteful, or better yet, nonexistent. Avoid large legal disclaimers tacked on to the end of every mail, especially when they express sentiments incompatible with participation in a free software project. For example, the following classic of the genre appears at the end of every post a particular user makes to a certain project mailing list:

> IMPORTANT NOTICE
>
> If you have received this e-mail in error or wish to read our e-mail disclaimer statement and monitoring policy, please refer to the statement below or contact the sender.
>
> This communication is from Deloitte & Touche LLP. Deloitte & Touche LLP is a limited liability partnership registered in England and Wales with registered number OC303675. A list of members' names is available for inspection at Stonecutter Court, 1 Stonecutter Street, London EC4A 4TR, United Kingdom, the firm's principal place of business and registered office. Deloitte & Touche LLP is authorised and regulated by the Financial Services Authority.
>
> This communication and any attachments contain information which is confidential and may also be privileged. It is for the exclusive use of the intended recipient(s).

> If you are not the intended recipient(s) please note that any form of disclosure, distribution, copying or use of this communication or the information in it or in any attachments is strictly prohibited and may be unlawful. If you have received this communication in error, please return it with the title "received in error" to IT.SECURITY.UK@deloitte.co.uk then delete the email and destroy any copies of it.
>
> E-mail communications cannot be guaranteed to be secure or error free, as information could be intercepted, corrupted, amended, lost, destroyed, arrive late or incomplete, or contain viruses. We do not accept liability for any such matters or their consequences. Anyone who communicates with us by e-mail is taken to accept the risks in doing so.
>
> When addressed to our clients, any opinions or advice contained in this e-mail and any attachments are subject to the terms and conditions expressed in the governing Deloitte & Touche LLP client engagement letter.
>
> Opinions, conclusions and other information in this e-mail and any attachments which do not relate to the official business of the firm are neither given nor endorsed by it.

For someone who's just showing up to ask a question now and then, that huge disclaimer looks a bit silly but probably doesn't do any lasting harm. However, if this person wanted to participate actively in the project, that legal boilerplate would start to have a more insidious effect. It would send at least two potentially destructive signals: first, that this person doesn't have full control over his tools — he's trapped inside some corporate mailer that tacks an annoying message to the end of every email, and he hasn't got any way to route around it — and second, that he has little or no organizational support for his free software activities. True, the organization has clearly not banned him outright from posting to public lists, but it has made his posts look distinctly unwelcoming, as though the risk of letting out confidential information must trump all other priorities.

If you work for an organization that insists on adding such signature blocks to all outgoing mail, and you can't get the policy changed, then consider using your personal email account to post, even if you're being paid by your employer for your participation in the project.

Avoiding Common Pitfalls

Certain anti-patterns appear again and again in threaded discussion forums. Below are the ones that seem to come up most often in open source project forums, and some advice on how to handle them.

Don't Post Without a Purpose

A common pitfall in online project participation is to think that you have to respond to everything. You don't. First of all, there will usually be more threads going on than you can keep track of, at least after the project really gets going. Second, even in the threads that you have decided to engage in, much of what people say will not require a response. Development forums in particular tend to be dominated by three kinds of messages:

1. Messages proposing something non-trivial
2. Messages expressing support or opposition to something someone else has said
3. Summing-up messages

None of these *inherently* requires a response, particularly if you can be fairly sure, based on watching the thread so far, that someone else is likely to say what you would have said anyway. (If you're worried that you'll be caught in a wait-wait loop because all the others are using this tactic too, don't be; there's almost always *someone* out there who'll feel like jumping into the fray.) A response should be motivated

by a definite purpose. Ask yourself first: do you know what you want to accomplish? And second: will it not get accomplished unless you say something?

Two good reasons to add your voice to a thread are a) when you see a flaw in a proposal and suspect that you're the only one who sees it, and b) when you see that miscommunication is happening between others, and know that you can fix it with a clarifying post. It's also generally fine to post just to thank someone for doing something, or to say "Me too!" if you want to strengthen a developing consensus, because a reader can tell right away that such posts do not require any response or further action, and therefore the mental effort demanded by the post ends cleanly when the reader reaches the last line of the mail. But even then, think twice before saying something; it's always better to leave people wishing you'd post more than wishing you'd post less. (The second half of Poul-Henning Kamp's "bikeshed" post, referenced from the section called “The Smaller the Topic, the Longer the Debate”, offers some further thoughts about how to behave on a busy mailing list.)

Productive vs Unproductive Threads

On a busy mailing list, you have two imperatives. One, obviously, is to figure out what you need to pay attention to and what you can ignore. The other is to behave in a way that avoids *causing* noise: not only do you want your own posts to have a high signal/noise ratio, you also want them to be the sorts of messages that stimulate *other* people to either post with a similarly high signal/noise ratio, or not post at all.

To see how to do that, let's consider the context in which it is done. What are some of the hallmarks of an unproductive thread?

- Arguments that have been made already start to be repeated in the same thread, as though the poster thinks no one heard them the first time.
- Increasing levels of hyperbole and involvement as the stakes get smaller and smaller.
- A majority of comments coming from people who do little or nothing, while the people who tend to get things done are silent.
- Many ideas discussed without clear proposals ever being made. (Of course, any interesting idea starts out as an imprecise vision; the important question is what direction it goes from there. Does the thread seem to be turning the vision into something more concrete, or is it spinning off into sub-visions, side-visions, and ontological disputes?)

Just because a thread is not productive at first doesn't mean it's a waste of time. It might be about an important topic, in which case the fact that it's not making any headway is all the more troublesome.

Guiding a thread toward usefulness without being pushy is an art. It won't work to simply admonish people to stop wasting their time, or to ask them not to post unless they have something constructive to say. You may, of course, think these things privately, but if you say them out loud then you will be offensive — and ineffective. Instead, you have to suggest conditions for further progress: give people a route, a path to follow that leads to the results you want, yet without sounding like you're dictating conduct. The distinction is largely one of tone. For example, this is bad:

> *This discussion is going nowhere. Can we please drop this topic until someone has a patch to implement one of these proposals? There's no reason to keep going around and around saying the same things. Code speaks louder than words, folks.*

Whereas this is good:

> *Several proposals have been floated in this thread, but none have had all the details fleshed out, at least not enough for an up-or-down vote. Yet we're also not saying anything new now; we're just reiterating what has been said before. So the best*

> *thing at this point would probably be for further posts to contain either a complete specification for the proposed behavior, or a patch. Then at least we'd have a definite action to take (i.e., get consensus on the specification, or apply and test the patch).*

Contrast the second approach with the first. The second way does not draw a line between you and the others, or accuse them of taking the discussion into a spiral. It talks about "we", which is important whether or not you actually participated in the thread before now, because it reminds everyone that even those who have been silent thus far still have a stake in the thread's outcome. It describes why the thread is going nowhere, but does so without pejoratives or judgements — it just dispassionately states some facts. Most importantly, it offers a positive course of action, so that instead of people feeling like discussion is being closed off (a restriction against which they can only be tempted to rebel), they will feel as if they're being offered a way to take the conversation to a more constructive level, if they're willing to make the effort. This is a standard the most productive people will naturally want to meet.

You won't always want a thread to make it to the next level of constructiveness — sometimes you'll want it to just go away. The purpose of your post, then, is to make it do one or the other. If you can tell from the way the thread has gone so far that no one is actually *going* to take the steps you suggested, then your post effectively shuts down the thread without seeming to do so. Of course, there isn't any foolproof way to shut down a thread, and even if there were, you wouldn't want to use it. But asking participants to either make visible progress or stop posting is perfectly defensible, if done diplomatically. Be wary of quashing threads prematurely, however. Some amount of speculative chatter can be productive, depending on the topic, and asking for it to be resolved too quickly will stifle the creative process, as well as make you look impatient.

Don't expect any thread to stop on a dime. There will probably still be a few posts after yours, either because mails got crossed in the pipe, or because people want to have the last word. This is nothing to worry about, and you don't need to post again. Just let the thread peter out, or not peter out, as the case may be. You can't have complete control; on the other hand, you can expect to have a statistically significant effect across many threads.

The Smaller the Topic, the Longer the Debate

Although discussion can meander in any topic, the probability of meandering goes up as the technical difficulty of the topic goes down. After all, the greater the technical complexity, the fewer participants can really follow what's going on. Those who can are likely to be the most experienced developers, who have already taken part in such discussions many times before, and know what sort of behavior is likely to lead to a consensus everyone can live with.

Thus, consensus is hardest to achieve in technical questions that are simple to understand and easy to have an opinion about, and in "soft" topics such as organization, publicity, funding, etc. People can participate in those arguments forever, because there are no qualifications necessary for doing so, no clear ways to decide (even afterward) if a decision was right or wrong, and because simply outwaiting other discussants is sometimes a successful tactic.

The principle that the amount of discussion is inversely proportional to the complexity of the topic has been around for a long time, and is known informally as the *Bikeshed Effect*. Here is Poul-Henning Kamp's explanation of it, from a now-famous post made to BSD developers:

> It's a long story, or rather it's an old story, but it is quite short actually. C. Northcote Parkinson wrote a book in the early 1960'ies, called "Parkinson's Law", which contains a lot of insight into the dynamics of management.
>
> [...]
>
> In the specific example involving the bike shed, the other vital component is an atomic power-plant, I guess that illustrates the age of the book.

> Parkinson shows how you can go in to the board of directors and get approval for building a multi-million or even billion dollar atomic power plant, but if you want to build a bike shed you will be tangled up in endless discussions.
>
> Parkinson explains that this is because an atomic plant is so vast, so expensive and so complicated that people cannot grasp it, and rather than try, they fall back on the assumption that somebody else checked all the details before it got this far. Richard P. Feynmann gives a couple of interesting, and very much to the point, examples relating to Los Alamos in his books.
>
> A bike shed on the other hand. Anyone can build one of those over a weekend, and still have time to watch the game on TV. So no matter how well prepared, no matter how reasonable you are with your proposal, somebody will seize the chance to show that he is doing his job, that he is paying attention, that he is *here*.
>
> In Denmark we call it "setting your fingerprint". It is about personal pride and prestige, it is about being able to point somewhere and say "There! *I* did that." It is a strong trait in politicians, but present in most people given the chance. Just think about footsteps in wet cement.

(Kamp's complete post is very much worth reading, too; see http://bikeshed.com/.)

Anyone who's ever taken regular part in group decision-making will recognize what Kamp is talking about. However, it is usually impossible to persuade *everyone* to avoid painting bikesheds. The best you can do is point out that the phenomenon exists, when you see it happening, and persuade the senior developers — the people whose posts carry the most weight — to drop their paintbrushes early, so at least they're not contributing to the noise. Bikeshed painting parties will never go away entirely, but you can make them shorter and less frequent by spreading an awareness of the phenomenon in the project's culture.

Avoid Holy Wars

A *holy war* is a dispute, often but not always over a relatively minor issue, which is not resolvable on the merits of the arguments, but where people feel passionate enough to continue arguing anyway in the hope that their side will prevail. Holy wars are not quite the same as bikeshed painting. People painting bikesheds may be quick to jump in with an opinion, but they won't necessarily feel strongly about it, and indeed will sometimes express other, incompatible opinions, to show that they understand all sides of the issue. In a holy war, on the other hand, understanding the other sides is a sign of weakness. In a holy war, everyone knows there is One Right Answer; they just don't agree on what it is.

Once a holy war has started, it generally cannot be resolved to everyone's satisfaction. It does no good to point out, in the midst of a holy war, that a holy war is going on. Everyone knows that already. Unfortunately, a common feature of holy wars is disagreement on the very question of *whether* the dispute is resolvable by continued discussion. Viewed from outside, it is clear that neither side is changing the other's mind. Viewed from inside, the other side is being obtuse and not thinking clearly, but they might come around if browbeaten enough. Now, I am *not* saying there's never a right side in a holy war. Sometimes there is — in the holy wars I've participated in, it's always been my side, of course. But it doesn't matter, because there's no algorithm for convincingly demonstrating that one side or the other is right.

A common, but unsatisfactory, way people try to resolve holy wars is to say "We've already spent far more time and energy discussing this than it's worth! Can we please just drop it?" There are two problems with this. First, that time and energy has already been spent and can never be recovered — the only question now is, how much *more* effort remains? If some people feel that just a little more discussion will resolve the issue in their favor, then it still makes sense (from their point of view) to continue.

The other problem with asking for the matter to be dropped is that this is often equivalent to allowing one side, the status quo, to declare victory by inaction. And in some cases, the status quo is known to be unacceptable anyway: everyone agrees that some decision must be made, some action taken. Dropping the subject would be worse for everyone than simply giving up the argument would be for anyone. But since that dilemma applies to all equally, it's still possible to end up arguing forever about what to do.

So how should you handle holy wars?

The first answer is, try to set things up so they don't happen. This is not as hopeless as it sounds:

You can anticipate certain standard holy wars: they tend to come up over programming languages, licenses (see the section called “The GPL and License Compatibility” in Chapter 9, *Legal Matters: Licenses, Copyrights, Trademarks and Patents*), reply-to munging (see the section called “The Great Reply-to Debate” in Chapter 3, *Technical Infrastructure*), and a few other topics. Each project usually has a holy war or two all its own, as well, which longtime developers will quickly become familiar with. The techniques for stopping holy wars, or at least limiting their damage, are pretty much the same everywhere. Even if you are positive your side is right, try to find *some* way to express sympathy and understanding for the points the other side is making. Often the problem in a holy war is that because each side has built its walls as high as possible and made it clear that any other opinion is sheer foolishness, the act of surrendering or changing one's mind becomes psychologically unbearable: it would be an admission not just of being wrong, but of having been *certain* and still being wrong. The way you can make this admission palatable for the other side is to express some uncertainty yourself — precisely by showing that you understand the arguments they are making and find them at least sensible, if not finally persuasive. Make a gesture that provides space for a reciprocal gesture, and usually the situation will improve. You are no more or less likely to get the technical result you wanted, but at least you can avoid unnecessary collateral damage to the project's morale.

When a holy war can't be avoided, decide early how much you care, and then be willing to publicly give up. When you do so, you can say that you're backing out because the holy war isn't worth it, but don't express any bitterness and *don't* take the opportunity for a last parting shot at the opposing side's arguments. Giving up is effective only when done gracefully.

Programming language holy wars are a bit of a special case, because they are often highly technical, yet many people feel qualified to take part in them, and the stakes are very high, since the result may determine what language a good portion of the project's code is written in. The best solution is to choose the language early, with buy-in from influential initial developers, and then defend it on the grounds that it's what you are all comfortable writing in, *not* on the grounds that it's better than some other language that could have been used instead. Never let the conversation degenerate into an academic comparison of programming languages (this seems to happen especially often when someone brings up Perl, for some reason); that's a death topic that you must simply refuse to be drawn into.

For more historical background on holy wars, see http://catb.org/~esr/jargon/html/H/holy-wars.html, and the paper by Danny Cohen that popularized the term, https://www.ietf.org/rfc/ien/ien137.txt.

The "Noisy Minority" Effect

In any mailing list discussion, it's easy for a small minority to give the impression that there is a great deal of dissent, by flooding the list with numerous lengthy emails. It's a bit like a filibuster, except that the illusion of widespread dissent is even more powerful, because it's divided across an arbitrary number of discrete posts and most people won't bother to keep track of who said what, when. They'll just have an instinctive impression that the topic is very controversial, and wait for the fuss to die down.

The best way to counteract this effect is to point it out very clearly and provide supporting evidence showing how small the actual number of dissenters is, compared to those in agreement. In order to

increase the disparity, you may want to privately poll people who have been mostly silent, but who you suspect would agree with the majority. Don't say anything that suggests the dissenters were deliberately trying to inflate the impression they were making. Chances are they weren't, and even if they were, there would be no strategic advantage to pointing it out. All you need do is show the actual numbers in a side-by-side comparison, and people will realize that their intuition of the situation does not match reality.

This advice doesn't just apply to issues with clear for-and-against positions. It applies to any discussion where a fuss is being made but it's not clear that most people consider the issue under discussion to be a real problem. After a while, if you agree that the issue is not worthy of action, and can see that it has failed to get much traction (even if it has generated a lot of mails), you can just observe publicly that it's not getting traction. If the "Noisy Minority" effect has been at work, your post will seem like a breath of fresh air. Most people's impression of the discussion up to that point will have been somewhat murky: "Huh, it sure feels like there's some big deal here, because there sure are a lot of posts, but I can't see any clear progress happening." By explaining how the form of the discussion made it appear more turbulent than it really was, you retrospectively give it a new shape, through which people can recast their understanding of what transpired.

Don't Bash Competing Open Source Products

Refrain from giving negative opinions about competing open source software. It's perfectly okay to give negative *facts* — that is, easily confirmable assertions of the sort often seen in good comparison charts. But negative characterizations of a less rigorous nature are best avoided, for two reasons. First, they are liable to start flame wars that detract from productive discussion. Second, and more importantly, some of the developers in *your* project may turn out to work on the competing project as well, or developers from the other project may be considering contributing in yours.

This kind of crossover is more likely than it at first might seem. The projects are already in the same domain (that's why they're in competition), and developers with expertise in a domain tend to make contributions wherever their expertise is applicable. Even when there is no direct developer overlap, it is likely that developers on your project are at least acquainted with developers on related projects. Their ability to maintain constructive personal ties could be hampered by overly negative marketing messages.

Bashing competing closed-source products seems to be more widely accepted in the open source world, especially when those products are made by Microsoft. Personally, I deplore this tendency (though again, there's nothing wrong with straightforward factual comparisons), not merely because it's rude, but also because it's dangerous for a project to start believing its own hype and thereby ignore the ways in which the competition may actually be superior. In general, watch out for the effect that marketing statements can have on your own development community. People may be so excited at being backed by marketing dollars that they lose objectivity about their software's true strengths and weaknesses. It is normal, and even expected, for a company's developers to exhibit a certain detachment toward marketing statements, even in public forums. Clearly, they should not come out and contradict the marketing message directly (unless it's actually wrong, though one hopes that sort of thing would have been caught earlier). But they may poke fun at it from time to time, as a way of bringing the rest of the development community back down to earth.

See also the related advice in the section called “Don't Bash Competing Vendors' Efforts” in Chapter 5, *Participating as a Business, Non-Profit, or Government Agency*.

Difficult People

Difficult people are no easier to deal with in electronic forums than they are in person. By "difficult" I don't mean "rude". Rude people are annoying, but they're not necessarily difficult. This book has already discussed how to handle them: comment on the rudeness the first time, and from then on, either

ignore them or treat them the same as anyone else. If they continue being rude, they will usually make themselves so unpopular as to have no influence on others in the project, so they are a self-containing problem.

The really difficult cases are people who are not overtly rude, but who manipulate or abuse the project's processes in a way that ends up costing other people time and energy, yet do not bring any benefit to the project[2].

Often, such people look for wedgepoints in the project's procedures, to give themselves more influence than they might otherwise have. This is much more insidious than mere rudeness, because neither the behavior nor the damage it causes is apparent to casual observers. A classic example is the filibuster, in which someone (always sounding as reasonable as possible, of course) keeps claiming that the matter under discussion is not ready for resolution, and offers more and more possible solutions, or new viewpoints on old solutions, when what is really going on is that he senses that a consensus or a ballot is about to form and he doesn't like where it's headed. Another example is when there's a debate that won't converge on consensus, but the group tries to at least clarify the points of disagreement and produce a summary for everyone to refer to from then on. The obstructionist, who knows the summary may lead to a result he doesn't like, will often try to delay even the summary, by relentlessly complicating the question of what should be in it, either by objecting to reasonable suggestions or by introducing unexpected new items.

Handling Difficult People

To counteract such behavior, it helps to understand the mentality of those who engage in it. People generally do not do it consciously. No one wakes up in the morning and says to himself: "Today I'm going to cynically manipulate procedural forms in order to be an irritating obstructionist." Instead, such actions are often preceded by a semi-paranoid feeling of being shut out of group interactions and decisions. The person feels he is not being taken seriously, or (in the more severe cases) that there is almost a conspiracy against him — that the other project members have decided to form an exclusive club, of which he is not a member. This then justifies, in his mind, taking rules literally and engaging in a formal manipulation of the project's procedures, in order to *make* everyone else take him seriously. In extreme cases, the person can even believe that he is fighting a lonely battle to save the project from itself.

It is the nature of such an attack from within that not everyone will notice it at the same time, and some people may not see it at all unless presented with very strong evidence. This means that neutralizing it can be quite a bit of work. It's not enough to persuade yourself that it's happening; you have to marshal enough evidence to persuade others too, and then you have to distribute that evidence in a thoughtful way.

Given that it's so much work to fight, it's often better just to tolerate it for a while. Think of it like a parasitic but mild disease: if it's not too debilitating, the project can afford to remain infected, and medicine might have harmful side effects. However, if it gets too damaging to tolerate, then it's time for action. Start gathering notes on the patterns you see. Make sure to include references to public archives — this is one of the reasons the project keeps records, so you might as well use them. Once you've got a good case built, start having private conversations with other project participants. Don't tell them what you've observed; instead, first ask them what they've observed. This may be your last chance to get unfiltered feedback about how others see the troublemaker's behavior; once you start openly talking about it, opinion will become polarized and no one will be able to remember what he formerly thought about the matter.

[2]For an extended discussion of one particular subspecies of difficult person, see Amy Hoy's hilariously on-target *http://slash7.com/2006/12/22/vampires/*. Quoting Hoy: "It's so regular you could set your watch by it. The decay of a community is just as predictable as the decay of certain stable nuclear isotopes. As soon as an open source project, language, or what-have-you achieves a certain notoriety — its half-life, if you will — *they* swarm in, seemingly draining the very life out of the community itself. *They* are the Help Vampires. And I'm here to stop them..."

If private discussions indicate that at least some others see the problem too, then it's time to do something. That's when you have to get *really* cautious, because it's very easy for this sort of person to try to make it appear as though you're picking on them unfairly. Whatever you do, never accuse them of maliciously abusing the project's procedures, of being paranoid, or, in general, of any of the other things that you suspect are probably true. Your strategy should be to look both more reasonable and more concerned with the overall welfare of the project, with the goal of either reforming the person's behavior, or getting them to go away permanently. Depending on the other developers, and your relationship with them, it may be advantageous to gather allies privately first. Or it may not; that might just create ill will behind the scenes, if people think you're engaging in an improper whispering campaign.

Remember that although the other person may be the one behaving destructively, *you* will be the one who appears destructive if you make a public charge that you can't back up. Be sure to have plenty of examples to demonstrate what you're saying, and say it as gently as possible while still being direct. You may not persuade the person in question, but that's okay as long as you persuade everyone else.

Case study

I remember only a few situations, in more than 20 years of working in free software, where things got so bad that we actually had to ask someone to stop posting altogether. In the example I'll use here, the person was not rude, and sincerely wanted only to be helpful. He just didn't know when to post and when not to post. Our lists were open to the public, and he was posting so often, and asking questions on so many different topics, that it was getting to be a noise problem for the community. We'd already tried asking him nicely to do a little more research for answers before posting, but that had no effect.

The strategy that finally worked is a perfect example of how to build a strong case on neutral, quantitative data. One of our developers, Brian Fitzpatrick, did some digging in the archives, and then sent the following message privately to a few developers. The offender (the third name on the list below, shown here as "J. Random") had very little history with the project, and had contributed no code or documentation. Yet he was the third most active poster on the mailing lists:

```
From: "Brian W. Fitzpatrick" <fitz@collab.net>
To: [... recipient list omitted for anonymity ...]
Subject: The Subversion Energy Sink
Date: Wed, 12 Nov 2003 23:37:47 -0600

In the last 25 days, the top 6 posters to the svn
[dev|users] list have been:

    294  kfogel@collab.net
    236  "C. Michael Pilato" <cmpilato@collab.net>
    220  "J. Random" <jrandom@problematic-poster.com>
    176  Branko #ibej <brane@xbc.nu>
    130  Philip Martin <philip@codematters.co.uk>
    126  Ben Collins-Sussman <sussman@collab.net>

I would say that five of these people are contributing to
Subversion hitting 1.0 in the near future.

I would also say that one of these people is consistently
drawing time and energy from the other 5, not to mention the
list as a whole, thus (albeit unintentionally) slowing the
development of Subversion.  I did not do a threaded
```

```
analysis, but vgrepping my Subversion mail spool tells me
that every mail from this person is responded to at least
once by at least 2 of the other 5 people on the above list.

I think some sort of radical intervention is necessary here,
even if we do scare the aforementioned person away.
Niceties and kindness have already proven to have no effect.

dev@subversion is a mailing list to facilitate development
of a version control system, not a group therapy session.

-Fitz, attempting to wade through three days of svn mail
 that he let pile up
```

Though it might not seem so at first, J. Random's behavior was a classic case of abusing project procedures. He wasn't doing something obvious like trying to filibuster a vote, but he was taking advantage of the mailing list's policy of relying on self-moderation by its members. We left it to each individual's judgement when to post and on what topics. Thus, we had no procedural recourse for dealing with someone who either did not have, or would not exercise, such judgement. There was no rule one could point to and say the fellow was violating it, yet everyone except him knew that his frequent posting was getting to be a serious problem.

Fitz's strategy was, in retrospect, masterful. He gathered damning quantitative evidence, but then distributed it discreetly, sending it first to a few people whose support would be key in any drastic action. They agreed that some sort of action was necessary, and in the end we called J. Random on the phone, described the problem to him directly, and asked him to simply stop posting. He never really did understand the reasons why; if he had been capable of understanding, he probably would have exercised appropriate judgement in the first place. But he agreed to stop posting, and the mailing lists became useable again. Part of the reason this strategy worked was, perhaps, the implicit threat that we could start restricting his posts via the moderation software normally used for preventing spam (see the section called “Spam Prevention” in Chapter 3, *Technical Infrastructure*). But the reason we were able to have that option in reserve was that Fitz had gathered the necessary support from key people first.

Handling Growth

The price of success is heavy in the open source world. As your software gets more popular, the number of people who show up looking for information increases dramatically, while the number of people able to provide information increases much more slowly. Furthermore, even if the ratio were evenly balanced, there is still a fundamental scalability problem with the way most open source projects handle communications. Consider mailing lists, for example. Most projects have a mailing list for general user questions — sometimes the list's name is "users", "discuss", "help", or something else. Whatever its name, the purpose of the list is always the same: to provide a place where people can get their questions answered, while others watch and (presumably) absorb knowledge from observing these exchanges.

These mailing lists work very well up to a few thousand users and/or a couple of hundred posts a day. But somewhere after that, the system starts to break down, because every subscriber sees every post; if the number of posts to the list begins to exceed what any individual reader can process in a day, the list becomes a burden to its members. Imagine, for instance, if Microsoft had such a mailing list for Windows. Windows has hundreds of millions of users; if even one-tenth of one percent of them had questions in a given twenty-four hour period, then this hypothetical list would get hundreds of thousands of posts per day! Such a list could never exist, of course, because no one would stay subscribed to it. This problem is not limited to mailing lists; the same logic applies to IRC channels, other discussion forums, indeed to any system in which a group hears questions from individuals. The implications are

ominous: the usual open source model of massively parallelized support simply does not scale to the levels needed for world domination.[3]

There will be no explosion when forums reach the breaking point. There is just a quiet negative feedback effect: people unsubscribe from the lists, or leave the IRC channel, or at any rate stop bothering to ask questions, because they can see they won't be heard in all the noise. As more and more people make this highly rational choice, the forum's activity will seem to stay at a manageable level. But it is staying manageable precisely because the rational (or at least, experienced) people have started looking elsewhere for information — while the inexperienced people stay behind and continue posting. In other words, one side effect of continuing to use unscalable communications models as a project grows is that the average *quality* of communications tends to go down. As the benefit/cost ratio of using high-population forums goes down, naturally those with the experience to do so start to look elsewhere for answers first. Adjusting communications mechanisms to cope with project growth therefore involves two related strategies:

1. Recognizing when particular parts of a forum are *not* suffering unbounded growth, even if the forum as a whole is, and separating those parts off into new, more specialized forums (i.e., don't let the good be dragged down by the bad).

2. Making sure there are many automated sources of information available, and that they are kept organized, up-to-date, and easy to find.

Strategy (1) is usually not too hard. Most projects start out with one main forum: a general discussion mailing list, on which feature ideas, design questions, and coding problems can all be hashed out. Everyone involved with the project is on the list. After a while, it usually becomes clear that the list has evolved into several distinct topic-based sublists. For example, some threads are clearly about development and design; others are user questions of the "How do I do X?" variety; maybe there's a third topic family centered around processing bug reports and enhancement requests; and so on. A given individual, of course, might participate in many different thread types, but the important thing is that there is not a lot of overlap between the types themselves. They could be divided into separate lists without causing harmful balkanization, because the threads rarely cross topic boundaries.

Actually doing this division is a two-step process. You create the new list (or IRC channel, or whatever it is to be), and then you spend whatever time is necessary gently nagging and reminding people to *use* the new forums appropriately. That latter step can last for weeks, but eventually people will get the idea. You simply have to make a point of always telling the sender when a post is sent to the wrong destination, and do so visibly, so that other people are encouraged to help out with routing. It's also useful to have a web page providing a guide to all the lists available; your responses can simply reference that web page and, as a bonus, the recipient may learn something about looking for guidelines before posting.

Strategy (2) is an ongoing process, lasting the lifetime of the project and involving many participants. Of course it is partly a matter of having up-to-date documentation (see the section called “Documentation” in Chapter 2, *Getting Started*) and making sure to point people there. But it is also much more than that; the sections that follow discuss this strategy in detail.

Conspicuous Use of Archives

Typically, all communications in an open source project, except sometimes IRC conversations, are archived. The archives are public and searchable, and have referential stability: that is, once a given piece of information is recorded at a particular address (URL), it stays at that address forever.

[3] An interesting experiment would be a probablistic mailing list, that sends each new thread-originating post to a random subset of subscribers, based on the approximate traffic level they signed up for, and keeps just that subset subscribed to the rest of the thread; such a forum could in theory scale without limit. If you try it, let me know how it works out.

Use those archives as much as possible, and as conspicuously as possible. Even when you know the answer to some question off the top of your head, if you think there's a reference in the archives that contains the answer, spend the time to dig it up and present it. Every time you do that in a publicly visible way, some people learn for the first time that the archives are there, and that searching in them can produce answers. Also, by referring to the archives instead of rewriting the advice, you reinforce the social norm against duplicating information. Why have the same answer in two different places? When the number of places it can be found is kept to a minimum, people who have found it before are more likely to remember what to search for to find it again. Well-placed references also contribute to the quality of search results in general, because they strengthen the targeted resource's ranking in Internet search engines.

There are times when duplicating information makes sense, however. For example, suppose there's a response already in the archives, not from you, saying:

```
It appears that your Scanley indexes have become frobnicated.  To
unfrobnicate them, run these steps:

1. Shut down the Scanley server.
2. Run the 'defrobnicate' program that ships with Scanley.
3. Start up the server.
```

Then, months later, you see another post indicating that someone's indexes have become frobnicated. You search the archives and come up with the old response above, but you realize it's missing some steps (perhaps by mistake, or perhaps because the software has changed since that post was written). The classiest way to handle this is to post a new, more complete set of instructions, and explicitly obsolete the old post by mentioning it:

```
It appears that your Scanley indexes have become frobnicated.  We
saw this problem back in July, and J. Random posted a solution at
http://blahblahblah/blah.  Below is a more complete description of
how to unfrobnicate your indexes, based on J. Random's instructions
but extending them a bit:

1. Shut down the Scanley server.
2. Become the user the Scanley server normally runs as.
3. As that user, run the 'defrobnicate' program on the indexes.
4. Run Scanley by hand to see if the indexes work now.
5. Restart the server.
```

(In an ideal world, it would be possible to attach a note to the old post, saying that there is newer information available and pointing to the new post. However, I don't know of any archiving software that offers an "obsoleted by" tag. This is another reason why creating dedicated web pages with answers to common questions is a good idea.)

Archives are probably most often searched for answers to technical questions, but their importance to the project goes well beyond that. If a project's formal guidelines are its statutory law, the archives are its common law: a record of all decisions made and how they were arrived at. In any recurring discussion, it's pretty much obligatory nowadays to start with an archive search. This allows you to begin the discussion with a summary of the current state of things, anticipate objections, prepare rebuttals, and possibly discover angles you hadn't thought of. Also, the other participants will *expect* you to have done an archive search. Even if the previous discussions went nowhere, you should include pointers to them when you re-raise the topic, so people can see for themselves a) that they went nowhere, and b) that you did your homework, and therefore are probably saying something now that has not been said before.

Treat All Resources Like Archives

All of the preceding advice applies to more than just mailing list archives. Having particular pieces of information at stable, conveniently findable addresses should be an organizing principle for all of the project's information. Let's take the project FAQ as a case study.

How do people use a FAQ?

1. They want to search in it for specific words and phrases.

   Therefore: the FAQ should be available in some sort of textual format.

2. They expect search engines such as Google to know about the FAQ's content, so that searches can result in FAQ entries.

   Therefore: the FAQ should be available as an HTML page.

3. They want to browse it, soaking up information without necessarily looking for answers to specific questions.

   Therefore: the FAQ should not only be available as an HTML page, it should be designed for easy browseability and have a table of contents.

4. They want to be able to refer other people directly to specific items in the FAQ.

   Therefore: each individual entry in the FAQ should be directly addresseable via a direct URL (e.g., using HTML IDs and named anchors, which are tags that allow people to reach a particular location on the page).

5. They want to be able to add new material to the FAQ, but note that this happens much less often than answers are looked up — FAQs are far more often read from than written to.

   Therefore: the source files for the FAQ should be conveniently available (see the section called "Version Everything" in Chapter 3, Technical Infrastructure), in a format that's easy to edit.

Formatting the FAQ like this is just one example of how to make a resource presentable. The same properties — direct searchability, availability to major Internet search engines, browsability, referential stability, and (where applicable) editability — apply to other web pages, to the source code tree, to the bug tracker, to Q&A forums, etc. It just happens that most mailing list archiving software long ago recognized the importance of these properties, which is why mailing lists tend to have this functionality natively, while other formats may require a little extra effort on the maintainer's part. Chapter 8, *Managing Participants* discusses how to spread that maintenance burden across many participants.

Codifying Tradition

As a project acquires history and complexity, the amount of data each new incoming participant must absorb increases. Those who have been with the project a long time were able to learn, and invent, the project's conventions as they went along. They will often not be consciously aware of what a huge body of tradition has accumulated, and may be surprised at how many missteps recent newcomers seem to make. Of course, the issue is not that the newcomers are of any lower quality than before; it's that they face a bigger acculturation burden than newcomers did in the past.

The traditions a project accumulates are as much about how to communicate and preserve information as they are about coding standards and other technical minutae. We've already looked at both sorts of standards, in the section called "Developer Documentation" in Chapter 2, *Getting Started* and the

section called “Writing It All Down” in Chapter 4, *Social and Political Infrastructure* respectively, and examples are given there. What this section is about is how to keep such guidelines up-to-date as the project evolves, especially guidelines about how communications are managed, because those are the ones that change the most as the project grows in size and complexity.

First, watch for patterns in how people get confused. If you see the same situations coming up over and over, especially with new participants, chances are there is a guideline that needs to be documented but isn't. Second, don't get tired of saying the same things over and over again, and don't *sound* like you're tired of saying them. You and other project veterans will have to repeat yourselves often; this is an inevitable side effect of the arrival of newcomers.

Every web page, every mailing list message, and every IRC channel should be considered advertising space — not for commercial advertisements, but for ads about your project's own resources. What you put in that space depends on the demographics of those likely to read it. An IRC channel for user questions, for example, is likely to get people who have never interacted with the project before — often someone who has just installed the software, and has a question he'd like answered immediately (after all, if it could wait, he'd have sent it to a mailing list instead, which would probably use less of his total time, although it would take longer for an answer to come back). Most people don't make a permanent investment in a support IRC channel; they'll show up, ask their question, and leave.

Therefore, the channel topic should be aimed at people looking for technical answers about the software *right now*, rather than at, say, people who might get involved with the project in a long term way and for whom community interaction guidelines might be more appropriate.

With mailing lists, the "ad space" is a tiny footer appended to every message. Most projects put subscription/unsubscription instructions there, and perhaps a pointer to the project's home page or FAQ page as well. You might think that anyone subscribed to the list would know where to find those things, and they probably do — but many more people than just subscribers see those mailing list messages. An archived post may be linked to from many places; indeed, some posts become so widely known that they eventually have more readers off the list than on it.

Formatting can make a big difference. For example, in the Subversion project, we were having limited success using the bug-filtering technique described in the section called “Pre-Filtering the Bug Tracker” in Chapter 3, *Technical Infrastructure*. Many bogus bug reports were still being filed by inexperienced people, and each time it happened, the filer had to be educated in exactly the same way as the 500 people before him. One day, after one of our developers had finally gotten to the end of his rope and flamed some poor user who didn't read the ticket tracker guidelines carefully enough, another developer decided this pattern had gone on long enough. He suggested that we reformat the ticket tracker front page so that the most important part, the injunction to discuss the bug on the mailing lists or IRC channels before filing, would stand out in huge, bold red letters, on a bright yellow background, centered prominently above everything else on the page. We did so (it's been reformatted a bit since then, but it's still very prominent — you can see the results at http://subversion.apache.org/reporting-issues.html), and the result was a noticeable drop in the rate of bogus ticket filings. The project still gets them, of course — it always will — but the rate has slowed considerably, even as the number of users increases. The outcome is not only that the bug database contains less junk, but that those who respond to ticket filings stay in a better mood, and are more likely to remain friendly when responding to one of the now-rare bogus filings. This improves both the project's image and the mental health of its participants.

The lesson for us was that merely writing up the guidelines was not enough. We also had to put them where they'd be seen by those who need them most, and format them in such a way that their status as introductory material would be immediately clear to people unfamiliar with the project.

Static web pages are not the only venue for advertising the project's customs. A certain amount of interactive monitoring (in the friendly-reminder sense, not the prison-panopticon sense) is also required. All peer review, even the commit reviews described in the section called “Practice Conspicuous

Code Review" in Chapter 2, *Getting Started*, should include review of people's conformance or non-conformance with project norms, especially with regard to communications conventions.

Another example from the Subversion project: we settled on a convention of "r12908" to mean "revision 12908 in the version control repository." The lower-case "r" prefix is easy to type, and because it's half the height of the digits, it makes an easily-recognizable block of text when combined with the digits. Of course, settling on the convention doesn't mean that everyone will begin using it consistently right away. Thus, when a commit mail comes in with a log message like this:

```
Patch from J. Random Contributor <jrcontrib@gmail.com>

* trunk/contrib/client-side/psvn/psvn.el:
  Fixed some typos from revision 12828.
```

...part of reviewing that commit is to say "By the way, please use 'r12828', not 'revision 12828' when referring to past changes." This isn't just pedantry; it's important as much for automatic parsability as for human readership.

By following the general principle that there should be canonical referral methods for common entities, and that these referral methods should be used consistently everywhere, the project in effect exports certain standards. Those standards enable people to write tools that present the project's communications in more useable ways — for example, a revision formatted as "r12828" could be transformed into a live link into the repository browsing system. This would be harder to do if the revision were written as "revision 12828", both because that form could be divided across a line break, and because it's less distinct (the word "revision" will often appear alone, and groups of numbers will often appear alone, whereas the combination "r12828" can only mean a revision number). Similar concerns apply to ticket numbers, FAQ items, etc.[4]

(Note that for Git commit IDs, the widely-accepted standard syntax is "`commit c03dd89305`, that is, the word "commit", followed by a space, followed by the first 8-10 characters of the commit hash. Some very busy projects have standardized on 12 characters, to avoid collisions; the only time all 40 characters of the hash are used is in non-human-readable contexts, like saving a commit ID in an automated release-tracking system or something.)

Even for entities where there is not an obvious short, canonical form, people should still be encouraged to provide key pieces of information consistently. For example, when referring to a mailing list message, don't just give the sender and subject; also give the archive URL *and* the Message-ID header. The last allows people who have their own copy of the mailing list (people sometimes keep offline copies, for example to use on a laptop while traveling) to unambiguously identify the right message in a search even if they don't have access to the archives. The sender and subject wouldn't be enough, because the same person might make several posts in the same thread, even on the same day.

The more a project grows, the more important this sort of consistency becomes. Consistency means that everywhere people look, they see the same patterns being followed, and start to follow those patterns themselves. This, in turn, reduces the number of questions they need to ask. The burden of having a million readers is no greater than that of having one; scalability problems start to arise only when a certain percentage of those readers ask questions. As a project grows, therefore, it must reduce that percentage by increasing the density and findability of information, so that any given person is more likely to find what he needs without having to ask.

Choose the Right Forum

[4] A more extended example of the kinds of benefits such standards make possible is the Contribulyzer example mentioned in the section called "The Automation Ratio" in Chapter 8, *Managing Participants*.

One of the trickiest things about managing an open source project is getting people to be thoughtful about which forum they choose for different kinds of communications. It's tricky partly because it's not immediately obvious that it matters. During any given conversation, the participants are mostly concerned with what the people involved are saying, and won't usually stop to think about whether or not the forum itself gives others who *might* want to take part the opportunity to do so.

For example, a real-time forum like IRC is terrific for quick questions, for opportunistic synchronization of work, for reminding someone of something they promised to do, etc. But it's not a good forum for making decisions that affect the whole project, because the people who take part in a conversation in IRC are just whoever happened to be in the channel at that moment — it's very dependent on work schedules, time zones, etc. On the other hand, the development mailing list is a great place for making formal project-wide decisions, since every interested party will have an opportunity to see and respond to the relevant posts, even though it's not as well-suited to quick, real-time interactions as IRC is.

Another example comes up frequently in bug tracker usage, especially in the last few years as bug trackers have become so well integrated with email. Sometimes people will be drawn into a discussion in a bug ticket[5] and because they simply see project-related emails coming in to their email client, they treat the discussion as though it's happening on the real development list. But it's not: anyone who wasn't watching that bug and wasn't explicitly invited into the conversation won't even be aware it's happening. If things are discussed in that bug ticket that go beyond the scope of just that one bug, they'll be discussed without input from people who should at least have had a chance to participate.

The solution to this is to encourage conscious, intentional forum changes. If a discussion starts to get into questions beyond the scope of its original forum, then at some point someone involved should ask that the conversation move over to the main development list or some other wider forum.

It's not enough for you to do this on your own. You have to create a culture where it's normal for everyone to do it, so everyone thinks about forum appropriateness as a matter of course, and feels comfortable raising questions of forum whenever necessary in any discussion. Obviously, documenting the practice will help (see the section called “Writing It All Down” in Chapter 4, *Social and Political Infrastructure*), but you'll probably also need to remind people of it often, especially when your project is starting out. A good rule of thumb is: if the conversation looks convergent, then it's okay to keep it in the bug ticket or other original forum. But if it looks likely to diverge for a while (e.g., widening into philosophical issues about how the software should behave, or raising design issues that go beyond just the one bug) before it converges, then take the discussion to a broader forum, usually the development mailing list.

Cross-Link Between Forums

When a discussion moves from one place to another, cross-link between the old and new place. For example, if discussion moves from the ticket tracker to the mailing list, link to the mailing list thread from the ticket, and mention the original ticket at the start of the new list thread. It's important for someone following the ticket to be able to reach the later discussion; it's also important for someone who encounters the ticket a year later to be able to follow to where the conversation went to in the mailing list archives. The person who does the move may find this cross-linking slightly laborious, but open source is fundamentally a writer-responsible culture. It's more important to make things easy for the tens or hundreds of people who may read the bug than for the three or five people writing about it.

It's also fine to take important conclusions or summaries from the list discussion and paste them into the ticket at the end, if that will make things convenient for readers. A common idiom is to move discussion to the mailing list, put a link to that thread in the ticket, and then when the discussion finishes, paste the final summary into the ticket (along with a link to the message containing that summary), so

[5]For example, on GitHub, simply mentioning someone's GitHub account name with an @-sign (e.g., `@kfogel`) in a comment on a ticket will cause that person to be added to the email thread associated with that ticket.

someone browsing the ticket later can easily see what conclusion was reached without having to click to somewhere else or do detective work. Note that the usual "two masters" data duplication problem does not exist here, because both archives and ticket comments are usually static, unchangeable data anyway.

Publicity

In free software, there is a fairly smooth continuum between purely internal discussions and public relations statements. This is partly because the target audience is always ill-defined: given that most or all posts are publicly accessible, the project doesn't have full control over the impression the world gets. Someone — say, a https://news.ycombinator.com/ poster or https://slashdot.org/ editor — may draw millions of readers' attention to a post that no one ever expected to be seen outside the project. This is a fact of life that all open source projects live with, but in practice, the risk is usually small. In general, the announcements that the project most wants publicized are the ones that will be most publicized, assuming you use the right mechanisms to indicate relative newsworthiness to the outside world.

Announcing Releases and Other Major Events

For major announcements, there tend to be a few main channels of distribution, on which announcements should be made as nearly simultaneously as possible:

1. Your project's front page is probably seen by more people than any other part of the project. If you have a really major announcement, put a blurb there. The blurb should be a very brief synopsis that links to the press release (see below) for more information.

2. At the same time, you should also have a "News" or "Press Releases" area of the web site, where the announcement can be written up in detail. Part of the purpose of a press release is to provide a single, canonical "announcement object" that other sites can link to, so make sure it is structured accordingly: either as one web page per release, as a discrete blog entry, or as some other kind of entity that can be linked to while still being kept distinct from other press releases in the same area.

3. Make sure the announcement gets broadcast by any relevant Twitter handles, and goes out on any news channels or RSS feeds. (The latter may happen automatically when you publish announcement, depending on how things are set up at your site.)

4. Post to forums as appropriate, in the manner described in the section called “Announcing”) in Chapter 2, *Getting Started*.

5. Send a mail to your project's announcement mailing list. This list's name should actually be "announce", that is, `announce@yourprojectdomain.org`, because that's a fairly standard convention now, and the list's charter should make it clear that it is very low-traffic, reserved for major project announcements. Most of those announcements will be about new releases of the software, but occasionally other events, such as a fundraising drive, the discovery of a security vulnerability (see the section called “Announcing Security Vulnerabilities”) later in this chapter, or a major shift in project direction may be posted there as well. Because it is low traffic and used only for important things, the `announce` list typically has the highest subscribership of any mailing list in the project (of course, this means you shouldn't abuse it — consider carefully before posting). To avoid random people making announcements, or worse, spam getting through, the `announce` list must always be moderated.

Try to make the announcements in all these places at the same time, as nearly as possible. People might get confused if they see an announcement on the mailing list but then don't see it reflected on the project's home page or in its press releases area. If you get the various changes (emails, web page edits, etc.) queued up and then send them all in a row, you can keep the window of inconsistency very small.

For a less important event, you can eliminate some or all of the above outlets. The event will still be noticed by the outside world in direct proportion to its importance. For example, while a new release of the software is a major event, merely setting the date of the next release, while still somewhat newsworthy, is not nearly as important as the release itself. Setting a date is worth an email to the daily mailing lists (not the announce list), and an update of the project's timeline or status web page, but no more.

However, you might still see that date appearing in discussions elsewhere on the Internet, wherever there are people interested in the project. People who are lurkers on your mailing lists, just listening and never saying anything, are not necessarily silent elsewhere. Word of mouth gives very broad distribution; you should count on it, and construct even minor announcements in such a way as to encourage accurate informal transmission. Specifically, posts that you expect to be quoted should have a clearly meant-to-be-quoted portion, just as though you were writing a formal press release. For example:

> *Just a progress update: we're planning to release version 2.0 of Scanley in mid-August 2005. You can always check http://www.scanley.org/status.html for updates. The major new feature will be regular-expression searches.*
>
> *Other new features include: ... There will also be various bugfixes, including: ...*

The first paragraph is short, gives the two most important pieces of information (the release date and the major new feature), and a URL to visit for further news. If that paragraph is the only thing that crosses someone's screen, you're still doing pretty well. The rest of the mail could be lost without affecting the gist of the content. Of course, sometimes people will link to the entire mail anyway, but just as often, they'll quote only a small part. Given that the latter is a possibility, you might as well make it easy for them, and in the bargain get some influence over what gets quoted.

Announcing Security Vulnerabilities

Handling a security vulnerability is different from handling any other kind of bug report. In free software, doing things openly and transparently is normally almost a religious credo. Every step of the standard bug-handling process is visible to all who care to watch: the arrival of the initial report, the ensuing discussion, and the eventual fix.

Security bugs are different. They can compromise users' data, and possibly users' entire computers. To discuss such a problem openly would be to advertise its existence to the entire world — including to all the parties who might make malicious use of the bug. Even merely committing a fix effectively announces the bug's existence (there are potential attackers who watch the commit logs of public projects, systematically looking for changes that indicate security problems in the pre-change code). Most open source projects have settled on approximately the same set of steps to handle this conflict between openness and secrecy, based on these basic guidelines:

1. Don't talk about the bug publicly until a fix is available; then supply the fix at the same time you announce the bug.

   It may make sense to supply the fix by packaging it as a release, or it may be enough to just commit it to the project's public repository. Whichever of those you you do, doing it effectively announces the vulnerability, so your formal announcement should go out in tandem with that fix.

2. Come up with that fix as fast as you can — especially if someone outside the project reported the bug, because then you know there's at least one person outside the project who is able to exploit the vulnerability.

In practice, those principles lead to a fairly standardized series of steps, which are described in the sections below.

Receive the Report

Obviously, a project needs the ability to receive security bug reports from anyone. But the regular bug reporting channels won't do, because they can be watched by anyone too. Therefore, have a separate mailing list or contact form for receiving security bug reports. That forum must not have publicly readable archives, and its subscribership must be strictly controlled — only long-time, trusted developers can be on the list, and people whom such developers have consensus that they trust[6]. If you need a formal definition of "trusted developer", you can use "anyone who has had commit access for two years or more" or something like that, to avoid favoritism. This is the group that will handle security bugs.

Ideally, that reporting gateway should not be spam-protected or moderated, since you don't want an important report to get filtered out or delayed just because no moderators happened to be online that weekend. If you do use automated spam-protection software, try to configure it with high-tolerance settings; it's better to let a few spams through than to miss a vulnerability report.

The submission mechanism should itself be secure. That is, if it is a contact form, it should be on an `https://` (TLS-protected) page, or if it is an email address, there should be a well-advertised public key (digitally signed by as many of the core developers as possible) so people can send encrypted mails to that address.[7] A web form submission or an email sent to your project may travel over many Internet hops on its way there; you have no reason to trust whoever runs those intermediate servers, and there is a flourishing market for new security vulnerabilities. Assume the worst and design accordingly.

Develop the Fix Quietly

So what does the security list do when it receives a report? The first task is to evaluate the problem's severity and urgency:

1. How serious is the vulnerability? Does it allow a malicious attacker to take over the computer of someone who uses your software? Or does it, say, merely leak information about the sizes of some of their files?

2. How easy is it to exploit the vulnerability? Can an attack be scripted, or does it require circumstantial knowledge, educated guessing, and luck?

3. *Who* reported the problem to you? The answer to this question doesn't change the nature of the vulnerability, of course, but it does give you an idea of how many other people might know about it. If the report comes from one of the project's own developers, you can breathe a little easier (but only a little), because you can trust them not to have told anyone else about it. On the other hand, if it came in an email from `anonymous14@globalhackerz.net`, then you'd better act as fast as you can. The person did you a favor by informing you of the problem at all, but you have no idea how many other people she's told, or how long she'll wait before exploiting the vulnerability on live installations.

Note that the difference we're talking about here is often just a narrow range between *urgent* and *extremely urgent*. Even when the report comes from a known, friendly source, there could be other

[6] E.g., a release manager who maybe isn't a core developer but who is already trusted to roll releases anyway. I've seen cases where companies who had been long involved in a project had managers as members of its security group, even though those managers had never committed a line of code, because by common consent the project's maintainers trusted them and felt it was to the project's benefit for them to see vulnerability reports as soon as possible. There is no one rule that will be appropriate for all projects, but in general, the core maintainers should follow the principle that anyone who receives security reports must be trustable both in terms of intention and in terms of their technical ability to not accidentally leak information (e.g., someone whose email gets hacked regularly should probably not be on the security list).

[7] If you don't know what all of these terms mean, find people you trust who do and get them to help your project. Handling security vulnerablities competently requiries a working knowledge of these concepts.

people on the Net who discovered the bug long ago and just haven't reported it. The only time things aren't urgent is when the bug inherently does not compromise security very severely.

The "`anonymous14@globalhackerz.net`" example is not facetious, by the way. You really may get bug reports from identity-cloaked people who, by their words and behavior, never quite clarify whether they're on your side or not. It doesn't matter: if they've reported the security hole to you, they'll feel they've done you a good turn, and you should respond in kind. Thank them for the report, give them a date on or before which you plan to release a public fix, and keep them in the loop. Sometimes they may give *you* a date — that is, an implicit threat to publicize the bug on a certain date, whether you're ready or not. This may feel like a bullying power play, but it's more likely a preëmptive action resulting from past disappointment with unresponsive software producers who didn't take security reports seriously enough. Either way, you can't afford to tick this person off. After all, if the bug is severe, she has knowledge that could cause your users big problems. Treat such reporters well, and hope that they treat you well.

Another frequent reporter of security bugs is the security professional, someone who audits code for a living and keeps up on the latest news of software vulnerabilities. These people usually have experience on both sides of the fence — they've both received and sent reports, probably more than most developers in your project have. They too will usually give a deadline for fixing a vulnerability before going public. The deadline may be somewhat negotiable, but that's up to the reporter; deadlines have become recognized among security professionals as pretty much the only reliable way to get organizations to address security problems promptly. So don't treat the deadline as rude; it's a time-honored tradition, and there are good reasons for it. Negotiate if you absolutely must, but remember that the reporter holds all the cards.

Once you know the severity and urgency, you can start working on a fix. There is sometimes a tradeoff between doing a fix elegantly and doing it speedily; this is why you must agree on the urgency before you start. Keep discussion of the fix restricted to the security list members, of course, plus the original reporter (if she wants to be involved) and any developers who need to be brought in for technical reasons.

Do not commit the fix to any public repository before the go-public date. If you were to commit it publicly, even with an innocent-looking log message, someone might notice and understand the change. You never know who is watching your repository and why they might be interested. Turning off commit emails wouldn't help; first of all, the gap in the commit mail sequence would itself look suspicious, and anyway, the data would still be in the repository. Just do all development in some private place known only to the people already aware of the bug.

CVE Numbers

You may have seen a *CVE number* associated with a particular security problems — e.g., a number like "CVE-2014-0160", where the first numeric part is the year, and the second is an increasing sequence number (it may exceed four digits if more than 10,000 numbers are handed out in a given year).

A CVE number is an entry in the "Common Vulnerabilities and Exposures" list maintained at https://cve.mitre.org/. The purpose of the list is to a provide standardized name for every known computer security problem, so that everyone has a unique, canonical name to use when discussing it, and a central place to go to find out more information.[8]

A CVE entry does not itself contain a full description of the bug and how to protect against it. Instead, it contains a brief summary, and a list of references to external resources (such as a announcement

[8] In the past, a CVE number would start out as a CAN number ("CAN" for "candidate") until it was approved for inclusion in the official list, at which point the "CAN" would be replaced with "CVE" while the number portion remained the same. However, nowadays they are just assigned a "CVE-" prefix from the start, although that prefix does not guarantee that the vulnerability will be included in the official list. (For example, it might be later discovered to be a duplicate of an existing CVE, in which case the earlier one — the lower number — should be used.)

post from the project in question) where people can go to get more detailed information. The real purpose of https://cve.mitre.org/ is to provide a well-organized space in which every vulnerability has a single name, and people have a clear route to get more data about it. See https://cve.mitre.org/cgi-bin/cvename.cgi?name=2014-0160 for an example of an entry. Note that the references can be very terse, with sources appearing as cryptic abbreviations. A key to those abbreviations is at https://cve.mitre.org/data/refs/refkey.html.

If your vulnerability meets the criteria, you may wish to obtain a CVE number for it. You can request one using the instructions at https://cve.mitre.org/cve/request_id.html, but if there is someone in your project who has already obtained CVE numbers, or who knows someone who has, let them do it. The CVE Editorial Board gets a lot of submissions, many of them spurious or poorly written submissions; by approaching them through a trusted source, you are saving them time and possibly getting your CVE number assigned more quickly. The other advantage of doing it this way is that somewhere along the chain, someone may know enough to tell you that a) it wouldn't count as a vulnerability or exposure according to MITRE's criteria, so there is no point submitting it, or b) the vulnerability already *has* a CVE number. The latter can happen if the bug has already been published on another security advisory list, for example at https://www.cert.org/ or on the BugTraq mailing list at http://www.securityfocus.com/. (If that happened without your project hearing about it, then you should worry what else might be going on that you don't know about.)

If you get a CVE number at all, you usually want to get it in the early stages of your bug investigation, so that all further communications can refer to that number. CVE entries are embargoed until the go-public date: the number will be reserved, but MITRE will allow some time (within reason) before revealing information about the vulnerability — so make sure you or your intermediary communicate clearly with the CVE Editorial Board about how long you need before publicly announcing.

See https://cve.mitre.org/ for more information about the CVE process. See also https://www.debian.org/security/cve-compatibility for a particularly clear exposition of one open source project's use of CVE numbers, and see https://securityblog.redhat.com/2013/01/30/a-minimal-security-response-process/ for a good writeup of a minimal security response process, from a security engineer at RedHat.

Common Vulnerability Scoring System (CVSS) Scores

Describing the severity of a vulnerability accurately is actually a difficult task. Does the vulnerability require physical access to the computer, or is network access enough? Does it require an authenticated user or not? Is it technically difficult to exploit, or can any bored teenager with some coding skills run it? Does it affect data integrity? Does it cause the software to crash?

Therefore, don't try to improvise language for expressing severity. Instead, use the *Common Vulnerability Scoring System* (*CVSS*) developed by the National Vulnerability Database at the U.S. National Institute of Standards:

https://nvd.nist.gov/cvss.cfm

NVD has thought very carefully about how to accurately and completely characterize severity for digital vulnerabilities, and their standardized expression format has become a standard in computer security. You can see an example in the "Severity:" section of the sample pre-notification email in the section called “Pre-Notification” below.

Pre-Notification

Once your security response team (that is, those developers who are on the security mailing list, or who have been brought in to deal with a particular report) has a fix ready, you need to decide how to distribute it.

If you simply commit the fix to your repository, or otherwise announce it to the world, you effectively force everyone using your software to upgrade immediately or risk being hacked. It is sometimes appropriate, therefore, to do *pre-notification* for certain important users.

Pre-notification is somewhat controversial, because it privileges some users over others. I personally think there are some circumstances where it is the right choice, particularly when there are well-known online services that use the software and that are tempting targets for attackers (perhaps because those services hold a lot of commercial or personal data about users). Those service's administrators would appreciate having an extra day or two to do the upgrade, so that they are already protected by the time the exploit becomes public knowledge — and their users, if they knew about this at all, would appreciate this too.

Pre-notification simply means contacting those administrators privately before the go-public date, telling them of the vulnerability and how to fix it. You should send pre-notification only to people you trust to be discreet with the information, and with whom you can communicate securely. That is, the qualification for receiving pre-notification is threefold: the recipient must run a large, important service where a compromise would be a serious matter; the recipient must be known to be someone who won't blab about the security problem before the go-public date; and you must have a way to communicate securely with the recipient, so that any eavesdroppers between you and your recipient can't read the message.[9]

Pre-notification should be done via secure means. If email, then encrypt it, for the same reasons explained in the section called “Receive the Report”, but if you have a phone number or other out-of-band secure way to contact the administrator, use that. When sending encrypted pre-notification emails, send them individually (one at a time) to each recipient. Do *not* send to the entire list of recipients at once, because then they would see each others' names — meaning that you would essentially be alerting each recipient to the fact that each *other* recipient may have a security hole in her service. Sending it to them all via blind CC (BCC) isn't a good solution either, because some admins protect their inboxes with spam filters that either block or reduce the priority of BCC'd mail, since so much spam is sent via BCC.

Here's a sample pre-notification mail:

```
From: Your Name Here
To: admin@large-famous-server.com
Reply-to: Your Name Here (not the security list's address)
Subject: Confidential important notification.

[[[ BEGIN ENCRYPTED AND DIGITALLY-SIGNED MAIL ]]]

This email is a confidential pre-notification of a security
alert in the Scanley server software.

Please *do not forward* any part of this mail to anyone.
The public announcement is not until May 19th, and we'd like
to keep the information embargoed until then.

You are receiving this mail because (we think) you run a
Scanley server, and would want to have it patched before
this security hole is made public on May 19th.

References:
```

[9]Remember that Subject lines in emails aren't encrypted, so don't put too much information about the vulnerability in a Subject line.

```
===========

   CVE-2017-892346: Scanley stack overflow in queries

Vulnerability:
==============

   The server can be made to run arbitrary commands if the
   server's locale is misconfigured and the client sends a
   malformed query.

Severity:
=========

   CVSSv2 Base Score: 9.0
   CVSSv2 Base Vector: AV:N/AC:L/Au:N/C:C/I:C/A:C

   (See https://nvd.nist.gov/CVSS/Vector-v2.aspx for how to
   interpret these expressions.)

Workarounds:
============

   Setting the 'natural-language-processing' option to 'off'
   in scanley.conf closes this vulnerability.

Patch:
======

   The patch below applies to Scanley 3.0, 3.1, and 3.2.

   A new public release (Scanley 3.2.1) will be made on or
   just before May 19th, so that it is available at the same
   time as this vulnerability is made public.  You can patch
   now, or just wait for the public release.  The only
   difference between 3.2 and 3.2.1 will be this patch.

[...patch goes here...]
```

If you have a CVE number, include it in the pre-notification (as shown above), even though the information is still embargoed and therefore the corresponding MITRE page will show nothing at the time of pre-notification. Including the CVE number allows the recipient to know with certainty that the bug they were pre-notified about is the same one they later hear about through public channels, so they don't have to worry whether further action is necessary or not, which is precisely the point of CVE numbers.

Distribute the Fix Publicly

The last step in handling a security bug is to distribute the fix publicly. In a single, comprehensive announcement, you should describe the problem, give the CVE number if any, describe how to work around it, and how to permanently fix it. Usually "fix" means upgrading to a new version of the software, though sometimes it can mean applying a patch, particularly if the software is normally run in source form anyway. If you do make a new release, it should differ from some existing release by exactly the security patch. That way, conservative admins can upgrade without worrying about what

else they might be affecting; they also don't have to worry about future upgrades, because the security fix will be in all future releases as a matter of course. (Details of release procedures are discussed in the section called “Security Releases” in Chapter 7, *Packaging, Releasing, and Daily Development*.)

Whether or not the public fix involves a new release, do the announcement with roughly the same priority as you would a new release: send a mail to the project's `announce` list, make a new press release, etc. While you should never try to play down the existence of a security bug out of concern for the project's reputation, you may certainly set the tone and prominence of a security announcement to match the actual severity of the problem. If the security hole is just a minor information exposure, not an exploit that allows the user's entire computer to be taken over, then it may not warrant a lot of fuss. See https://cve.mitre.org/about/terminology.html for a good introduction to how to think about and discuss vulnerabilities.

In general, if you're unsure how to treat a security problem, find someone with experience and talk to them about it. Assessing and handling vulnerabilities is very much an acquired skill, and it's easy to make missteps the first few times.

See also the Apache Software Foundation guidelines on handling security vulnerabilities at https://www.apache.org/security/committers.html. They are an excellent checklist you can compare against to see if you're doing everything carefully.