acmqueue

A Nice Piece of Code

Colorful metaphors and properly reusing functions

George V. Neville-Neil

In the last installment of Kode Vicious (A System is not a Product, *ACM Queue* 10 (4), April 2012), I mentioned that I had recently read two pieces of code that had actually lowered, rather than raised, my blood pressure. As promised, this edition's KV covers that second piece of code.

One of the things that can make me *really* like a piece of code—other than the obvious ones such as decent documentation, proper indenting, and rational variable names—is when a function or subsystem is properly reused. Over the past month, I've been reading the IPFW (IP Firewall) code (written by Luigi Rizzo at the University of Pisa), which is one of the firewalls available in FreeBSD. Like any firewall, IPFW needs to examine packets and then decide to drop, modify, or pass the packets unchanged through the system. Having reviewed several pieces of software that do similar work, I have to say that IPFW does the best job of reusing the code around it. Here are two examples.

Part of the job of a firewall is to classify packets and then decide what to do with them. There are a few ways to go about this, but what IPFW does is quite elegant. It reuses a tried and tested idea from another place in the kernel, the BPF (Berkeley Packet Filter). BPF classifies packets using a set of opcodes—sort of like a machine language for processing network packet headers—to decide whether a packet matches a filter that has been specified by the user. Using opcodes and a state machine for packet classification leads to a flexible and compact implementation of the packet classifier, compared with hand-coding rules for later use. IPFW extends the set of opcodes that can be used for classifying packets, but the idea is exactly the same, and the resulting code is easy to read and understand, and therefore easier to maintain and less likely to contain bugs that might let malicious packets through. The entire state machine that executes any and all firewall rules in IPFW is only 1,200 lines of C code, including comments. Another advantage of using a set of opcodes to express the packet-processing rules is that the entire chunk of C code, which is really a bytecode interpreter, can be replaced by just-in-time compiled code, generated by an optimizing compiler. This leads to an even greater increase in packet-processing speed.

A more direct case of reuse is how IPFW directly reuses the kernel's routing-table code to store its own address lookup tables. Many of the rules in a firewall make reference to the source or destination address of a packet. While it is quite possible to write your own routines for storing and retrieving network addresses—and many people have—there is no need to rewrite this code, in particular when your firewall code will already be linked into a program that has such routines available. The radix code in the kernel can manage any type of key/value lookup, although it is optimized to handle network addresses and associated masks. The IPFW table-management code is really just a simple wrapper around the radix code, as can be seen in the following lookup code:

```

1  int
2  ipfw_lookup_table(struct ip_fw_chain *ch, uint16_t tbl, in_addr_t addr,
   uint32_t *val)
3  {
4      struct radix_node_head *rn timer;
5      struct table_entry *ent;
6      struct sockaddr_in sa;
7
8      if (tbl >= IPFW_TABLES_MAX)
9          return (0);
10     rn timer = ch->tables[tbl];
11     KEY_LEN(sa) = 8;
12     sa.sin_addr.s_addr = addr;
13     ent = (struct table_entry *) (rn timer->rn timer_lookup(&sa, NULL, rn timer));
14     if (ent != NULL) {
15         *val = ent->value;
16         return (1);
17     }
18     return (0);
19 }

```

All this code does is take arguments understood by IPFW, such as the chain of rules (ch), address table (tbl), and address being sought (addr), and pack them up in a way that is usable by the radix code, which is called on line 13. The value is returned in the last argument to the function. All of the other functions in the table-management code, which add, delete, and list entries in the table, look very much the same. They are wrappers around the radix code. Treating the routing-table code as a library, as IPFW does, means writing less complex and tedious code, and results in a mere 200 lines of C code, including comments, to implement tables of network addresses. It is this sort of reuse, not the tortured kind that I more often come across, that leads me to praise this code.

Don't worry, I'm sure next time I'll be back to ranting about some bad bit of code, but I have to say it has been a nice surprise to have found two well-written pieces of code in two months. I think it's some sort of record.

KV

Dear KV,

One of the people on my current project keeps complaining about my use of "colorful metaphors" in code. While I understand that I shouldn't be checking these sorts of things into our source repo, I can't see why he complains when he sees them on my screen. I mostly use such words for debugging messages because they're shocking enough to stand out from the rest of the log messages produced by our software. I can't really believe that KV would object to a programmer adding a bit of salt to log messages.

Kolorful Koder

Dear Kolorful,

I can understand why you might think that I might be a prolific user of colorful metaphors, given some of the things I write about in this column. You are correct, and my coworkers can tell you that, because of my occasional outbursts when dealing with particularly horrific bits of code, they have learned a thing or two about body parts and functions that they wish they could now forget.

Unfortunately, for you at least, I have to come down on the side of your coworker in this dispute. While I'm sure you have faithfully marked every place that your code might exclaim a colorful metaphor with the well-known comment, "XXX Remove This!", the fact is that if you do this enough, someday, and usually on quite the wrong day, you're going to forget. You probably think you won't, but the risk is not worth the eventual hassle. I've been through that hassle, and I'm glad that, for once, the problem wasn't my fault.

More than a decade ago I worked for a company that produced a software IDE (integrated development environment) and some associated low-level software. One of the IDE's limitations, on a certain platform, was that every project saved by the IDE had to have an appropriate extension—those letters after the dot—that provide a clue about what type of file has just been saved. While programmers are quite used to giving their files such descriptive monikers as `notes.txt`, `main.c`, and `stdlib.h`, it turns out that not everyone is familiar with this sort of naming standard, and some even prefer names such as `Project1` and `Project2`, without any type of identifying extension.

The programmer working on the IDE decided that if the user of his program declined to add an extension to the project file name, he would add one for them. He chose a four-letter word that rhymes with duck. I'm not sure if he meant this to go out in the release, as a way of pointing out customers who refused to use file extensions, or if it was something he meant to change before the release, but in the end it didn't matter. Within days of our 1.0.1 maintenance release of the IDE, there was a 1.0.1b release with a single change. I don't remember if the b release had a note saying what changed, but all of the engineers working on the software knew the real reason.

Amazingly, the programmer who did this got to keep his job. I suspect there were two reasons for this, the first being that he was actually a pretty good programmer, and the second being that he was the only one in the company willing to support the IDE on the platform that he was working on.

While this is a pretty extreme example of a colorful metaphor gone wrong, and while I know that there are programmers who will leave extremely strong language in comments, I have to say that I frown on this as well. Your code is your legacy, and while your mother might never see it, you should still only check in code that would not shock her should she choose to read it.

KV

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who currently lives in New York City.

© 2012 ACM 1542-7730/11/0600 \$10.00