Code Spelunking

Exploring Cavernous Code Bases

GEORGE V. NEVILLE-NEIL,
CONSULTANT

Try to remember your first day at your first software job. Do you recall what you were asked to do, after the human resources people were done with you? Were you asked to write a piece of fresh code? Probably not. It is far more likely that you were asked to fix a bug, or several, and to try to understand a large, poorly documented collection of source code.

Of course, this doesn't just happen to new graduates; it happens to all of us whenever we start a new job or look at a new piece of code. With experience we all develop a set of techniques for working with large, unfamiliar source bases. This is what I call *code spelunking*.

Code spelunking is very different from other engineering practices because it is done long after the initial design and implementation of a system. It is a set of forensic techniques used after the crime has been committed.

There are several questions that code spelunkers need to ask, and tools are available to help them answer these questions. I will look at some of these tools, addressing their shortcomings and pointing out possible improvements.

The background is a complex, abstract composition. It features a dark blue to black color palette. Overlaid on this are various geometric elements: sharp, angular shapes in lighter shades of blue and white, a fine grid pattern, and a prominent white DNA double helix structure on the left side. The overall effect is one of high-tech complexity and digital abstraction.

Code diving through
unfamiliar source bases is something
we do far more often than
write new code from scratch—
make sure you have
the right gear for the job.

Code Spelunking

Exploring Cavernous Code Bases

SIZE DOES MATTER

Source bases are already large, and getting larger all the time. Table 1 shows the sizes of some popular open source software. The Linux kernel, with support for 17 different processor architectures, is made up of 642 directories, 12,417 files, and more than 5 million lines of code. A complex network server such as Apache is made up of 28 directories and 471 files—encompassing over 158,000 lines of code—whereas an application such as the `nvi` editor contains 29 directories, 265 files, and over 77,000 lines of code. I believe that these examples are fairly honest representations of what people are confronted with when they start a job.

Of course, even larger systems exist for scientific, military, and financial applications, but those shown in table 1 are more familiar and should help convey an instinctive feeling for the complexity involved in systems that we all come in contact with every day.

Unfortunately, the tools we have to work with often lag behind the code we are trying to explore. There are several reasons for this, but the primary one is that very few companies have been able to build a thriving business on tools. It is much easier to make money selling software that appeals to a broad audience, which software tools do not.

TABLE 1

Sizes of Popular Software

Program	Version	Directories	Files	Lines
Apache Web Server	1.3	28	471	158,332
DB	1.4.25	176	234	99,836
Emacs	21	43	2586	1,317,915
FreeBSD Kernel	5.1	420	4758	2,140,517
Linux Kernel	2.4.20-8	642	12,417	5,223,290
Nvi	1.81.5	29	265	77,176
Python	2.2.3	245	1158	356,314

Note: Every attempt was made to disregard documentation or other files not directly related to the source code.

DOWN THE RABBIT HOLE (AKA PATHS TO TAKE)

Static vs. dynamic. We can use two scales to compare code spelunking tools and techniques. The first one ranges from static analysis at one end to dynamic analysis at the other. In static analysis you're not observing a running program but are examining only the source code. A clear example is using the tools `find`, `grep`, and `wc` to give you a better idea of the size of the source code (as was done to produce table 1). A code review is an example of a static technique. You print the code, sit down at a table with it, and begin to read.

On the dynamic end of the scale are tools such as debuggers. Here you run the code on real data, using one program (the debugger) to examine another as it executes. Another dynamic tool is a software oscilloscope, which displays a multithreaded program as a series of horizontal time lines—like those on a real oscilloscope—to find deadlocks, priority inversions, and other bugs common to multithreaded programs. Oscilloscopes are used mostly in embedded systems.

Brute force vs. subtlety. The second type of scale measures the level of finesse that is applied to spelunking. At one extreme is a brute-force approach, which often consumes a lot of CPU time or which might generate a large amount of data. An example of brute force is attempting to find a bug by using `grep` to locate a message that prints out near to where the error occurs.

At the other end of the finesse scale is subtlety. A subtle approach to finding a string within a program involves a tool that builds a database of all the interesting components of a code base (e.g., function name, structure definitions, and the like). You then use that tool to generate a fresh database each time you update your source from the code repository. You would also use it when you

want to know something about the source, as it already has the information you need at its virtual fingertips.

Plotting your method.

You can use both of these scales to create a two-dimensional graph in which the x-axis represents finesse and the y-axis runs from static to dynamic. You can then plot tools and techniques

on this graph so that they can be compared (see figure 1). None of the terms used implies value judgments on the techniques. Sometimes a static, brute-force approach is just what you want. If it would take just as long (or longer) to set up a tool to do some subtle analysis, a brute-force approach makes sense. You don't often get points for style when code spelunking; people just want results.

One thing to keep in mind when code spelunking is the Scout motto, "Be prepared." A little effort spent up front getting your tools in place will always save time in the long run. After a while, all engineers develop a stable of tools that they use to solve problems. I have a few tools that I always keep on hand when code spelunking, and I make sure to prepare the ground with them whenever I start a project. These tools include Cscope and global, which I'll discuss in more detail later.

NEEDLE IN A HAYSTACK (DEBUGGING)

A very common scenario in code spelunking is attempting to fix a bug in an unfamiliar piece of code. Debugging is a highly focused task: You have a program, it runs, but not correctly. You must find out why it does this, where it does this, and then repair it. What's wrong with the program is usually your only known quantity. Finding the needle buried in the haystack is your job, so the first question must be, "Where does the program make a mistake?"

You can approach the problem in several ways, many of which are shown in figure 1. The approach you choose depends on the situation. If the program is a single file, you could probably find the bug through inspection, but as table 1 demonstrates, any truly useful application is much larger than a single file.

Let's take a theoretical example. Jack takes a job with the Whizzo Company, which produces WhizzoWatcher, a media player application that can play and decode many types of entertainment content. On his first day at work (after he has signed up for health insurance, the stock plan, and 401K) Jack's boss e-mails him two bug reports to deal with.

The two bugs that Jack has just been assigned are as follows:

- Bug 1: When WhizzoWatcher opens a file of type X it immediately crashes with no output except a core file.
- Bug 2: While watching a long movie on DVD (*The Lord of the Rings, The Two Towers*) the audio sync is lost after about two hours. This does not happen at any particular frame; it varies.

WhizzoWatcher 1.0 is a typically organic piece of software. Originally conceived as a "prototype," it wowed the VPs and investors and was immediately rushed into production over the objections of the engineers who had written it. It has little or no design documentation, and what exists is generally inaccurate and out of date. The only real source of information on the program is the code itself. Because this was a prototype, several pieces of open source software were integrated into the larger whole and were expected to be replaced when the "real system" was funded. The total system now consists of about 500 files that spread over 15 directories, some of which were written in-house and some of which were integrated.

Bug 1. This is the easiest bug for Jack to work on; the program crashes when it starts. He can run it in the debugger, and the offending line will be found at the next crash. In terms of code spelunking, he doesn't have to look at much of the code at first, although becoming generally familiar with the code base will help him in the long run.

Unfortunately, Jack finds that although the reason for the immediate crash is obvious, what caused it is not. A common cause of crashes in C code is dereferencing a null pointer. At this point in the debugger, Jack doesn't have the previous states of the program, only the state at the moment it crashed, which is actually a very small amount of data. A common technique is to visually

Code Spelunking

Exploring Cavernous Code Bases

inspect the code while stepping up the stack trace to see if some caller stomped on the pointer.

A debugger that could step backward, even over a small number of instructions, would be a boon in this situation. On entry into a function the debugger would create enough storage for all of the function's local variables and the application's global variables so that they could be recorded, statement by statement, throughout the function. When the debugger stopped (or the program crashed), it would be possible to step backward up to the start of the function to find out which statement caused the error. For now, Jack is left to read the code and hope to stumble across the real cause of the error.

Bug 2. Attacking Bug 2, where the code doesn't crash but only produces incorrect results, is more difficult because there isn't a trivial way to get a debugger to show you when to stop the program and inspect it. If Jack's debugger supports conditional breakpoints and watchpoints, then these are his next line of defense. A watchpoint or a conditional breakpoint tells the debugger to stop when something untoward happens to a variable, and allows Jack to inspect the code at a point closest to where the problem occurs.

Once Jack has found the problem, it's time to fix it. The key is to fix the bug without breaking anything else in the system. A thorough round of testing is one way to address this problem, but he'll feel more comfortable making a fix if he can find out more about what it affects in the system. Jack's next question ought to be, "Which routines call the routine I want to fix?"

Attempting to answer this question with the debugger will not work, because Jack cannot tell from the debugger all the sources that will call the offending routine. This is where a subtle, static approach bears fruit. The tool Cscope builds a database out of a chunk of code and allows him to do the following:

1. Find a C symbol
2. Find a global definition
3. Find functions called by a function
4. Find functions calling a function

5. Find a text string
6. Change a text string
7. Find an egrep pattern
8. Find a file
9. Find all files that #include this file

You will note that item 4, "Find functions calling a function," answers his question. If the routine he's about to fix modifies nonlocal variables or structures, he must next answer the question, "With which functions does my function share data?" This would never come up in a "properly designed program" (i.e., one written from scratch). Jack would never use a horde of global variables to control his program, because he knows what a nightmare it would be to maintain. Right?

Of course, this is code spelunking, so Jack is already past that point. Using a subtle tool such as Cscope again is tempting, but this turns out to be a case where brute force is his best hope. Generating a list of all the global references in a program (file name and line number) is certainly possible for a compiler, but none of them do this. Although this option would hardly be used in the creation of new code, it would make the process of debugging old code far easier. Jack will have to make do with a combination of find and grep to figure out where all these variables reside in the program.

SECURING YOUR CAVE (SPELUNKING FOR SECURITY)

Code spelunking isn't something you do only when debugging; it's how you perform a good code review, reverse-engineer a design document, and conduct a security audit.

During an age when many people use computers for financial and personal transactions, auditing code for security holes has (or should) become commonplace. In order to close these security holes, you have to know what the common attacks are, as well as which sections of the code are vulnerable to attack. Although the attacks are updated almost daily on the Bugtraq mailing list (<http://www.securityfocus.com>), what we're concerned with is finding them.

Consider the following example. Jill takes a job with a large bank that serves most of its customers electronically, over the Internet. On her first day her boss tells her that the bank is doing a security audit of its entire system, which is implemented on several different types of hardware and operating systems. One set of Unix servers handles incoming Web traffic and then makes requests to a mainframe backend that actually manages the real money.

One possible security hole occurs when a program stores a user's private data in a way that makes it available to anyone with access to the machine. An example of this is storing a password as plain text in a file or registry key. If Jill had written the software, or had access to the person who did, she could quickly find out where the program stores passwords simply by asking. Unfortunately, the person who wrote the login scripts was laid off when the bank moved its headquarters six months ago. Jill will have to find out how this is done without the author's help.

Unlike in the debugging case, few tools are available to tell Jill something as specific as, "Where does the program store data X?" Depending on the structure of the program, this could happen anywhere—and often does. Jill can tackle this problem with either a brute-force or

a subtle approach. To do the former she would litter the code with debugging statements (i.e., `printfs` or language equivalent) to find out where the program stores the password. She probably has a few guesses to narrow the field from the entire program down to a few or a dozen locations, but this is still a good deal of work.

A more subtle approach would be to run the program in the debugger, attempt to stop the program right after the user enters a password, and then follow the execution until the program performs the storage operation.

A typical way to attack a program is to give it bad input. To find these vulnerabilities, Jill must ask the question, "Where does the program read data from an untrustworthy source?" Most people would immediately think that any code dealing with network data would be vulnerable, and they would be right—in part. With the

TOOL RESOURCES

Global

<http://www.gnu.org/software/global/>

This is the tool I apply to every source base that I can. Global is really a pair of tools: `gtags` and `htags`. The first, `gtags`, builds a database of interesting connections based on a source tree in C, C++, Java, or Yacc. Once the database is built, you can use your editor (both Emacs and vi are supported) to jump around within the source. Want to know where the function you're calling is defined? Just jump to it. The second tool is `htags`, which takes the source code and the database generated by `gtags` and creates an HTML-browsable version of the source code in a subdirectory. This means that, even if you don't use Emacs or vi, you can easily jump around the source code finding interesting connections. Building the database is relatively speedy, even for large code bases, and should be done each time you update your sources from your source-code control system.

Cscope

<http://cscope.sourceforge.net/>

Cscope was originally written at AT&T Bell Labs in the 1970s. It answers many questions, such as: Where is this symbol? Where is something globally defined? Where are all the functions called by this function? Where are all the functions that call this function? Like Global, it first builds a database from your source code. You can then use a command-line tool, Emacs, or one of a few GUIs written to work with the system to get your questions answered.

gprof

<http://www.gnu.org/manual/gprof-2.9.1/gprof.html>

This is the standard profiling tool on most open source Unix systems. Its output is a list of routines ordered by how often they were called and how much of the program's CPU time was spent executing them. This can be useful in figuring out where to start looking for security holes in a program.

ktrace

This is a standard tool on open source operating systems. The name stands for "kernel trace." It will give you a listing of all the system calls made by a program. The output includes the arguments and return values given to and received from the calls. You use `ktrace` by running a program under it and then dumping the output with `kdump`. It is available on all open source Unix operating systems.

truss

<http://www.sun.com/software/solaris/programs/abi/>

This is available only on Solaris. It is the Solaris version of `ktrace`—basically the same tool, with different arguments.

code spelunking

<http://www.codespelunking.org>

This Web site contains a TWiki collaboration page dedicated to code spelunking.

Code Spelunking

Exploring Cavernous Code Bases

advent of networked file systems, the fact that the code is executing a `read()` statement does not imply that it is reading from a local (i.e., trustable) source.

Whereas Jack's debugging in my earlier example attempted to zero in on a problem, Jill's code audit is more of a "fan-out." She wants to see as much of the code as possible and understand how it interacts both with its own modules ("How is data passed around?") and with external entities ("Where do we read and write data?"). This can be an even more daunting task than finding a needle in a haystack, and may be more like labeling all the individual straws. In this case, the most important thing for Jill to do is to find the most likely places that will cause problems—that is, the places that are executed most often.

There is a tool that, although not originally intended for this purpose, can help to focus Jill's efforts: a code profiler. For example, `gprof` was originally written to tell an engineer which routines within the program are using up all the CPU time and are, therefore, candidates for optimization. The program is run with a workload (let a user bang on it or have it service requests from the network), and then the output is analyzed. A profiler will tell Jill which routines are being called most often. These are obviously the routines to check first. There is no reason for Jill to pore over the large portion of the code that is called infrequently while the routines called most often may have gaping holes.

Routines that pass bad arguments to system calls are another common security problem. This is most often exploited against network servers in an effort to gain control over a remote machine. Several commercial tools attempt to discover these kinds of problems. A quick and dirty approach is to use a system call tracer, such as `ktrace` or `truss`, to make a record of what system calls are executed and what their arguments are. This can give you a good idea of where possible errors lie.

SOMETIMES THE BEST TOOL IS YOU

Code spelunking is about asking questions. The challenge is to get your fingers around the right parts of the code

and find the answers without having to look at every line (which is a near impossibility anyway). There is one tool I haven't yet mentioned in this article, and that's the one sitting inside your head. You can begin applying good engineering practices even though they weren't used to create the code you're spelunking.

Keeping a journal of notes on what you've found and how things work will help you to create, and keep in mind, a picture of how the software you're exploring works. Drawing pictures is also a great help; these should be stored electronically along with your notes. Good experiment design of the type you may have learned—and then promptly forgotten—in physics class is also helpful. Just beating on a piece of code until it works is not an efficient way of figuring it out. Setting up an experiment to see why the code behaves a certain way may take a lot of thought, but usually very little code.

Finally, one of my favorite tools is the "stupid programmer trick." You get a colleague to look at the code and then you attempt to explain to him or her what the code does. Nine times out of ten your colleague won't even need to say anything. Fairly quickly you realize what's going on, smack yourself on the forehead, say thank you, and go back to work. Through the process of systematically explaining the code out loud, you have figured out the problem.

There is no one tool that will make understanding large code bases easier, but I hope that I've shown you a few ways to approach this task. I suspect you'll find more on your own. Q

LOVE IT, HATE IT? LET US KNOW:

queue-ed@acm.org or www.acmqueue.org/forums

GEORGE V. NEVILLE-NEIL has worked in the embedded systems area for the past eight years, both as an integrator of final products and as an implementor of off-the-shelf embedded operating systems. His work has centered on the networking aspects of embedded systems, but he has also done general work on broader aspects of the systems. Neville-Neil developed a device-driver model for networking devices used in VxWorks, worked on a multi-instance version of the Berkeley TCP/IP stack, and ported open source networking code to VxWorks. He is currently working on a new, commercial, dynamic host configuration protocol (DHCP) server at Nominum. He also teaches seminars and classes. His computer-related interests are networking, operating systems, embedded systems, and software componentization.

© 2003 ACM 1542-7730/03/0900 \$5.00.