Silver Bullet Talks with Bart Miller

Gary McGraw | Cigital

Hear the full podcast at www.computer.org/silverbullet. Show links, notes, and an online discussion can be found at www.cigital.com/silverbullet.

Bart Miller, professor of computer science at the University of Wisconsin-Madison and chief scientist of the US Department of Homeland Security’s Software Assurance Marketplace (SWAMP) research facility, discusses Heartbleed, fuzz testing, his work on dynamic instrumentation of binaries, and the SWAMP project.

One of my favorite papers about Heartbleed was the one that you wrote with James Kupsch. Tell us about the methods you describe for software assurance and how they worked or didn’t work against the OpenSSL code base. Heartbleed was a wake-up call for a lot of people who were making assumptions about the security of open source software. It was also a wake-up call for people who were depending on software assurance tools to scan and look for flaws in code. Why didn’t software assurance tools, well-known names like Fortify, Coverity, CodeSonar, AppScan, or Red Lizard, find this vulnerability? We were surprised that these tools missed a simple buffer overflow.

One of the issues, of course, is that when code is a disastrous mess from a control- or dataflow perspective, it makes the job harder for some of those tools. It definitely does. There’s a lot of complexity in the code, so we tried to study it carefully from the perspective of knowing how tools work because we’ve worked on tools ourselves, and knowing how exploits are written because we’ve written our own exploits. We found a lot of pointers and aliasing, with different pointers to data structures meaning different things. Reuse of data structures was big, too—just an interesting layering of code complexity. However, the open source community has a lot of great things going for it. In fact, in our early fuzz testing, we determined that overall code reliability was better in the open source world than in commercial code by far.

Just in terms of fuzzing—dynamic analysis told you that. That’s right. There are many good things about open source software. The problem is, to paraphrase [executive chairman of Google] Eric Schmidt, there’s no adult supervision: you have a lot of very smart people, but nobody coming in and saying that’s just too clever or too cute. Code complexity really hampered the ability of these tools to find the vulnerability, and you can see that same cleverness in the fix for that vulnerability. It has this if statement, and if you look at that if statement, anybody with any amount of software background will cringe. It says if 1 plus 2 plus 16 is greater than something, then... . You’re just burying magic numbers in your code. It’s incredible. I teach an undergrad OS course, and I make my students check all the return code from their system calls, or they can’t turn their program in. They can’t have magic constants buried in their code—basic, basic stuff. While the fix is definitely a fix, it’s indicative of a need for more discipline in the software engineering process.
One of the things that we find in the BSIMM [Building Security In Maturity Model] is people using fuzzing, so to speak, on APIs inside a piece of code. You have a component that has an API, so you build a grammar that you can then use for fuzzing. Does that fit your definition of the term? I think it’s cool that fuzzing has grown much bigger than anything we’ve ever done. [Editors’ note: Miller founded the field of fuzz random testing in the 1980s; http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html.] We did a simple version of what you described—what some people call structured fuzzing—when testing the very first Windows-based program. We tried a variety of different things, the first one being to send packets, in this case, random junky packet streams—byte streams—to see what would happen. We had a few crashes because people tended to use libraries that don’t send random unstructured packet streams, but when we looked at the control flow, we realized that we were catching all the buffering and formatting, not the deeper operations. So we said, well, let’s do fuzzing, which basically meant hammering away at a keyboard and mouse randomly. A few years later, the Linux guys built a tool that would actually test system and function calls, and they basically fuzzed the kernel call interface.

Was that Valgrind?
No. Valgrind checks for memory errors in application programs. This is actually just randomly calling the live C functions that involve kernel calls and using random parameters in each of them—it’s generous to call it a grammar; it’s a specification for each parameter of its range and how many parameters are where.

You’ve done a lot of work with Jeff Hollingsworth on dynamic instrumentation of binaries. What aspects of this work are interesting or informative for security?
Our goal was to look at a binary program, parse it, and analyze it, much like a compiler would look at a source program. That’s a fairly conventional notion now, but in the 1990s, it was really just us and the Digital Equipment Corp. looking at it seriously. The goal was also to patch a binary program on the fly while it was running. People thought that was crazy, which translates into a good research project. The idea was to be able to analyze binary code, do a deep control- and dataflow analysis of it, structurally understand the program, be able to control the program, and then patch it safely while it’s running.

We’ve had two interesting applications in security and cyberforensics. One watches malware because malware is defensive. It unpacks itself, it manipulates its code on the fly, it uses pointer-based control flow, and it uses obscure instruction sequences. We built an open source version of our binary instrumentation tools, called Dyninst, that watched code execute—anytime you couldn’t tell what the program was doing, it would instrument it and catch the action. Anytime new code was unpacked or modified, we caught it before it happened and watched. We could analyze and instrument any piece of code before it executed.

We also started using advanced machine-learning techniques to see if we could look at binary code and tell which language each module was written in, which compiler, which optimization level, which version of the compiler—all those things were pretty straightforward. So then we wondered, could we tell the code author by looking at the binary? We all know programmers have their style. If you’ve worked in a group for a while, you can say, “Oh, that’s Larry’s code. That’s Susie’s code.” You know people’s style. So our hypothesis was that the authorship signal, that style, would be maintained through the compiling, through the optimization, through the linking process, through stripping of all the symbols and all the symbolic names, and we found good, strong authorship signals in binary code.

In geopolitical terms, it’d be interesting to see if there was Russian code versus Filipino code versus Chinese code in the malware pile.
In fact, that question is very interesting to analysts because bad guys assemble modules from well-defined marketplaces, your sled code, your implant, your exfil code—if you could look at each of those modules and figure out an author assignment, you can start drawing social networks of bad actors and actually start tracing back sources and who’s working with whom. You can also use it for more mundane things like intellectual property rights arguments.

About Bart Miller
Bart Miller is a professor of computer science at the University of Wisconsin-Madison. He’s also chief scientist of the US Department of Homeland Security–funded Software Assurance Marketplace, fondly known as SWAMP. Miller received a PhD from the University of California, Berkeley. He’s an ACM Fellow and coined the term “fuzz testing.” His research encompasses binary code analysis, computer security, and software testing, among other disciplines, and he lives in Madison, Wisconsin.

www.computer.org/security
You’re the chief scientist of the SWAMP project; how does it compare with existing commercial solutions?
A big difference is that you don’t pay any money for it; the goal is to bring together a large collection of software assurance tools. We’re starting with static analysis tools that are doing source and bytecode, and we’ll move on to various dynamic tools, mobile tools, Web tools, and so on.

Are you using commercial tools inside SWAMP?
We’re starting with all the open source tools, and we’re in discussions with several commercial vendors—until we get to release their tools, I can’t say which ones. But the basic idea is we go to these software assurance meetings and look at 10 different tools, open source or commercial, and they are all good at different things. We want to offer not a single instrument but a symphony. The idea is to take the results from a large collection of tools; automatically get your application program to run against them; and get a nice, clean, merged result.

Because our application packages aren’t just applications but are also reference packages, you as a tool writer can tune your instrument. The typical SWAMP user can bring her software package to SWAMP and run it against a collection of software analysis tools. You can run it against known code bases. You can run it against unknown code bases and compare the results to other tools.

Let’s say you’re an assistant professor. You have a really cool idea, and you want to develop an analysis that does one good thing, but you can’t build a general-purpose tool—nobody’s going to use it. If you bring it into SWAMP and add it to the analysis, that one good thing you do will be better for every SWAMP user. We allow people to make incremental contributions without having to build the entire complex infrastructure they need to build a tool that somebody else will use. We also want to make researchers happy. We want to bring students and classes in; we have some high school students who are using SWAMP.

What makes computer security different from other kinds of software testing?
I think it’s the fact that we have an active, malicious adversary. In software testing, we’re victims of our own carelessness, schedules, lack of skill, or too much skill, but in security, there are adversaries out there with infinite time and resources. I say infinite resources because there may be tens of thousands of hobby hackers out there who are smart and who have infinite time; infinite resources could also mean well-funded bad guys. Testing is a safety mechanism whereas security is really a battleground.

Last question, which has nothing to do with anything: SCUBA or skiing?
I’ve often thought about a way of combining those, but I never came up with it.

You really should. There should be a way to ski down a cliff and then land in a lake with all your gear on. Until that day, skiing gets the slight nod.

The Silver Bullet Podcast with Gary McGraw is cosponsored by Cigital and this magazine and is syndicated by SearchSecurity.

Gary McGraw is Cigital’s chief technology officer. He’s the author of Software Security: Building Security In (Addison-Wesley 2006) and eight other books. McGraw has a BA in philosophy from the University of Virginia and a dual PhD in computer science and cognitive science from Indiana University. Contact him at gem@cigital.com.