Foreword

Software is easy to criticize and hard to do. The bigger the software, the more that is true. It is thus like speech—the more you say, the easier it is for the reader to find something to criticize, and the more likely the critic will get it wrong. Brevity may be the soul of wit, but it is wit that is the soul of brevity.

And, indeed, our software is nothing if not loquacious, slang-riven, ill-bred, bloated, and raw. Is it any wonder that software is as prone to misinterpretation as is our language, any wonder that our software, like our language, can be "twisted by knaves to make a trap for fools?" No, it is not, but, as with language, everything we collectively are now depends on software. Software is so very essential that it is unlikely that the world's population would be as great as it now is without software—software to transport, to transact, to transcribe, to translate, to transmit, to transform. In other words, the evidence is unarguable that we have to get software right, just as the evidence is unarguable that getting software right does not, and will not, come naturally.

As Dr. McGraw reminds us, breaking something is easier than designing something that cannot be broken, though I personally prefer Sam Rayburn's earthy formulation, *viz.*: "Any jackass can kick down a barn, but it takes a good carpenter to build one." And that is what makes secure software in particular the pinnacle of concern because the very definition of secure software is that it withstands sentient opponents. Parsing that definition in its contrapositive: If a product does not have sentient opponents, then it does not have security requirements. This is best examined by looking at why products fail—if your product fails because of a collection of clueless users ("Hey, watch this!"), alpha particles, or discharged batteries, security is not your issue. If your product fails because some gleeful clown discovers that he can be the super-user by typing 5000 lowercase As into some prompt, said clown may not be all that sentient, but nevertheless your product has security requirements.

This can't be a completely bright line, but it is an instructive distinction. Secure software is, by definition, designed with failure in mind. Secure software resists failure even when that failure is devoutly wished for by the opponent. Secure software is designed for the failure case as much as or more than the success case. Designers and implementers alike envision an opponent who can think.

As Dr. McGraw says throughout this book, baking in security only happens when there is intent to do so. My father used to scold me when my excuse for this or that was "I didn't mean to do it, Daddy." His stinging comeback, for which I am a better man, was always "But did you mean not to?" Given what I do for a living, I read vulnerability reports every day. Every one of them says, "I didn't mean to do it, Daddy." Sometimes they even try to say, "I didn't do it, but if I did I didn't mean to, and anyway you didn't notice, so all you have to do is install this tiny little fix unless you want what happens next to be your fault; aren't I a good boy?" I want to scream "Did you mean not to?" even though the honest answer will at best be "I thought I meant not to."

There is not enough security expertise to go around. Good people are hard to find, and the need for them rises faster than the supply of them. What do you do when some skill is rare but needful? You convert rare expertise into a process that others can follow, but the kind of process has to be one that reinforces disciplined thinking, avoids patronizing the people on whom it must be imposed, and can be measured sufficiently well to know if it works. Better still if the process is one where you don't have to take all or nothing, where you can get real value out of doing only some of it. Better to do it all, but at the limit any process will have diminishing return so partial value for partial effort is a good thing. Dr. McGraw, describing himself as not naturally a process person, does exactly what I asked for above.

A good idea is one where, once you've heard it, you say, "Well, that's obvious." Much of what you will find in this book has that quality—you will be tempted to say, "Well, that's obvious." For example, the idea that code review is the highest power weapon you can train on software security. For example, that you can't know how much of a fight your software will have to put up when challenged unless you study hard how it might come under intentional abuse. Of course, the process is only good if you use it. Buffer overflows remain the most common attack method, and we've known how to avoid them for years, so knowing what to do is provably insufficient.

You might say, “What makes Cigital’s process better than XYZ’s process?” For that there is one clear logical response: The question is moot. There is so little effective being done that there must be something wrong. That “something wrong” is either a shortage of skill or a shortage of discipline. If it is a shortage of skill, experts are duty bound to share what works in a way that others can use. There may be many workable processes, but this book shows there is at least one. With this book, the clock is ticking; any continuing failure must trace to a shortage of discipline. We’ll know soon enough.

If the reader would prefer some numbers even in the Foreword, here are three: There’s a new Windows virus every four hours. Perhaps 15% of all desktop machines are running malware of some sort. Embedded systems outnumber desktop machines by between one and two orders of magnitude, and they are almost never field upgradeable. The *raison d’être* for this book is thus shown useful.

My own research has satisfied me that the spread between the firms with the best software security practices and the worst is growing wider; my best guess is a disparity (measured by ratios of flaw density between best and worst) that is doubling every twelve months. If you believe, as I and Dr. McGraw do, that security is a subset of reliability, you have merely to borrow availability calculus: With five systems components in an e-commerce application, each of which has 98% uptime, you should expect to be down 2.5 hours per day.

Security is to software what mutation is to natural selection, but with the overwhelmingly important difference: With software security you are in control of your survival advantage. If that sounds attractive, adopt at least some of the McGraw/Cigital program. It won’t be easy and it won’t be fun, but as the U.S. Army Ranger Handbook says:

> *Two of the gravest general dangers to survival are the desire for comfort and a passive outlook.*

Ball’s in your court.

Dan Geer
September 17, 2005
Cambridge, MA