Food for Thought

Improving the Market for Assurance

Carl E. Landwehr
Editor in chief

Wisconsin's milk market had a problem in the 1880s: some farmers produced milk with higher butterfat content than others, but because it was difficult for sellers and buyers to measure it, the lower-quality milk brought the same price as the higher-quality. This inequity encouraged practices such as watering the milk, and it held back the industry as a whole.

Happily, a University of Wisconsin professor, Stephen M. Babcock, invented a relatively simple and inexpensive process for determining the butterfat content of milk, and he made it available without patent. This invention enabled the market for milk to function better, allowing consumers to reward high-quality milk producers and avoid low-quality ones. The industry thrived, and today Wisconsin is known as "America's Dairyland." The story recurred in 1970s India, and the country has subsequently become a leading milk producer.

This example shows how better information can improve a marketplace. George Akerlof's famous paper on "The Market for 'Lemons'" argues that asymmetrical information (that is, the seller knows more than the buyer about the offered product)—such as was present prior to the ability to measure milk fat—can easily cause a market to decline.

Today's market for software also exhibits information asymmetry. The security properties of a piece of software are hard to specify and still harder to assure. Though producers might not know precisely how trustworthy their products are, they can know their ingredients—the sources of the software and hardware, the competence of the individuals involved, the assurance procedures used, and the time and effort invested.

But consumers, lacking this information, have difficulty establishing whether one product is less vulnerable than another, so it's difficult for them to reward stronger products in the marketplace. The available measurement tools are either crude and ineffective (as with many of the checklists applied in system-certification exercises) or complex to apply (as in the Common Criteria evaluation process). Both of these approaches are labor-intensive, hence costly, and must be reapplied as systems change.

Can we imagine a tool that a modern day Babcock might develop that could improve the security marketplace?

Although many sources of vulnerability exist in our computer systems today, the ones that continue to provide the largest source of exploitations are fundamentally programming errors—errors of implementation or relatively low-level design, such as unchecked buffers and unvalidated inputs that enable stack smashing and cross-site scripting attacks. In the past 20 years, considerable progress has been made in developing tools that can detect such errors at the source-code level and increasingly even at the object-code level. Type checkers, taint checkers, model checkers, and verifiers are among the tools now available for this purpose.

We've also developed programming languages that make it impossible to commit broad classes of errors. Java and C# are well-known examples, with their strong typing, which prevents buffer overflows and a wide variety of other code attacks.

Unfortunately, the assurance processes and procedures that are in use in the world at large don't generally make systematic use of these tools to provide any kind of guarantee about the code that winds up running on our systems. The Common Criteria evaluation processes don't even require direct examination of the source code until you reach Evaluation Assurance Level (EAL) 4—the highest level generally applied to commercial products.

An interesting question is, using mechanical means alone, how much assurance can we get that a software component is free of a reasonably broad set of vulnerabilities? We'll never be sure there are no residual vulnerabilities, of course, but our current processes don't provide that assurance, either. Couldn't we imagine a much less labor-intensive, yet more effective, approach to assuring that our software makes exploitations difficult?

A clear answer to this question might require some assumptions or

Letters

To the Editors:

We have found two flaws in the fundamental argument of "Alien vs. Quine" by Vanessa Gratzer and David Naccache in the March/April 2007 issue. One is mathematical, and one is based on systems engineering.

The fundamental argument of the article appears to be that the Quine program *Q* is the smallest possible program that, when supplied with random input *R*, can output the string *QR*. But we know from Turing that any claims about the "smallest possible program" must be considered very carefully—such claims can only be proven by iterating through all possible smaller programs and showing that none of them produce the desired result.

As *R* gets larger, this is harder and harder to prove. For a large *R*—as might be necessary with a computer on which we might wish to run a security-critical function—it is possible that a piece of malware could analyze *R* and find a sequence of bytes in which it might be able to embed portions of itself. This analysis could be performed as the string was written to memory. The chance of the malware being successful increases as the size of *R* increases. For a large *R*, we think it is quite likely that a hostile program could accept *R* as input and output *QR*.

From a systems' approach, Gratzer and Naccache assume that they completely understand the processor's instruction set and the computer's architecture. Our experience doesn't support this view. Real-world systems are riddled with undocumented instructions, hidden registers, and so on.

The authors ask, "Can this approach be adapted to more complex and modern architectures?" We feel the answer is "no." As systems become more complex and as their memories become larger, the chances of this system working appear to decrease.

—George Dinolt and Simson Garfinkel, US Naval Postgraduate School

Gratzer and Naccache reply:

There are no flaws in our paper—Dinolt and Garfinkel simply misunderstood it.

Dinolt and Garfinkel write that "such claims can only be proven by…," but we do not claim any proofs for space-constrained Quines. Moreover, we heavily insist that a proof couldn't be found ("…we can't provide a formal proof…") and illustrate the difficulty in crafting such proofs by a concrete example (`Quine3.asm`). This difficulty is precisely why we introduce the time-constrained Quine, whose behavior we do prove through time measurements.

Dinolt and Garfinkel "think it is quite likely" that a hostile program could exist. The challenge set by our paper is not that of thinking that such programs exist, nor even proving their existence, but that of exhibiting (actually constructing) them: "security practitioners can proceed by analogy to cryptosystems whose specifications are published and subject to public scrutiny. If we find an *M* simulating *Q* with respect to φ, a fix can either replace *Q*, φ, or both." A question on which Dinolt and Garfinkel's letter offers no progress.

We all know that collisions in hash functions provably exist, but that exhibiting collisions is very hard. Similarly, we challenged the readers to exhibit malware that is always successful in simulating our Quine (any failure probability < 1 can be amplified by iteration). The statement "If we find an *M* simulating *Q*" clearly shows that we don't rule out the existence of simulators. We do, however, conjecture that exhibiting simulators is hard. If Dinolt and Garfinkel think that crafting always-successful polynomial-time Quine simulators is easy, they are welcome to exhibit one.

Dinolt and Garfinkel state that real-world systems may have hidden instructions. We assume that hardware is known ("given a state machine μ") and that the software isn't. It is a standard practice in security research to define adversarial models and reason within them, and we don't claim results in unknown hardware settings. That being said, we still fail to grasp the novelty in the "hidden-instruction flaw statement": hidden instructions can potentially threaten any security software we can think of. What rules out the existence of maliciously hidden Pentium instructions that recognize undocumented binary sequences and covertly start leaking hard disk contents via email? Do we declare PGP "flawed" because we can't rule-out this possibility? In what way are our Quines different?

Finally, a word on the "feel"ing that our approach cannot be adapted to modern architectures.

Security research history taught us that whenever the community felt that a new idea didn't suit existing machines, one of two things happened: either the idea died, or machines were modified to meet it.

When RSA was invented many certainly "felt" that adding 1024-bit multipliers to microprocessors was a folly. Today, microprocessors with 1024-bit multipliers abound. We do not pretend that our idea scales with RSA, but it is our role as researchers to invent, imagine, and dare. And, why not, even hope the advent of "assembly language [modified] to allow such proofs"… □

constraints on software or system development processes. For example, we might be able to support a claim that there are no buffer-overflow vulnerabilities in a piece of source code, either by examining the code mechanically or writing the software in a language in which such errors are impossible to make. Either way, we could have a system that could simply be recompiled or automatically reanalyzed, with a minimum of human labor, when changes were made.

An evaluation/certification process that leveraged modern programming languages and analytic tools could accelerate both the development and the adoption of less vulnerable and more effective programming practices, products, and systems. □