Why Attacking Systems Is a Good Idea

Iván Arce
Core Security Technologies

Gary McGraw
Cigital

When Dan Farmer, then of Silicon Graphics and Wietse Venema of Eindhoven University of Technology, decided to publicly release the network scanning tool SATAN (Security Administrator Tool for Analyzing Networks) way back in 1995, he was fired for refusing to abandon a project that could be used to attack systems. Nine years later, every system administrator uses network scanning tools as a regular part of their job. The logic behind improving your network's security posture by breaking into it is no longer questioned. (You can find Farmer and Venema's classic paper, "Improving the Security of Your Site by Breaking Into It," at http://nsi.org/Library/Compsec/farmer.txt.) In an ironic twist of fate, administrators who don't use tools like Satan on a regular basis now run the risk of being fired.

Yet when University of Calgary professor John Aycock announced a course on malware in Fall 2003, many indignant security vendors (including Trend Micro and Sophos) lined up to criticize him. The vendors claim that teaching students to understand and create malicious code is a mistake—that no good could come of such a course. On the flip side, the justification for such a course lies in the idea that the motivations and mechanisms of malicious code must be understood in order to be properly combated.

So who's right? Should we talk about attacking systems? Should we teach people how real attacks work? Is ethical hacking an oxymoron? Those are the sorts of questions that motivate this special issue of *IEEE Security & Privacy*.

A popular opinion?

The security and privacy research community appears to be mostly united on the issue of discussing attacks in the open—the majority opinion is that the only way to properly defend a system against attack is to understand what attacks look like as deeply and realistically as possible. Unless we discuss attacks explicitly—publishing articles and books about attacks, giving presentations about system vulnerabilities, and so on—we will be practicing security by obscurity, or, worse yet, security by merely crossing our fingers.

The fact is, system security is difficult to get right. Short of building provably secure systems (which some dream of, but is nonetheless an activity that seems not to be done with any regularity), a critical part of security design and implementation is subjecting a system to rigorous analysis, including attack-based penetration testing. Strangely, this means that designers must understand what makes attackers tick so as not to fall prey to this kind of testing.

It's important to emphasize that none of the information we discuss in this special issue is news to the malicious hacker community. Some security experts might worry that revealing the techniques described in articles like these will encourage more people to try them out—an echo of the worry that got Dan Farmer fired. Perhaps this is true, but cybercriminals and hooligans have always had better lines of communication and information-sharing than the good guys. We believe that security professionals must understand and digest this kind of information so that they grasp the magnitude of the problem and begin to address it properly.

Article diversity

The articles in this issue represent the broad range of ideas that come to mind when scientists and engineers think about attacking systems. Some approach the problem by describing technically sophisticated attacks, attack patterns, and toolkits. Others fret about the politics of security and attacks, worrying that outlawing certain kinds of software will backfire. Some describe methodologies for

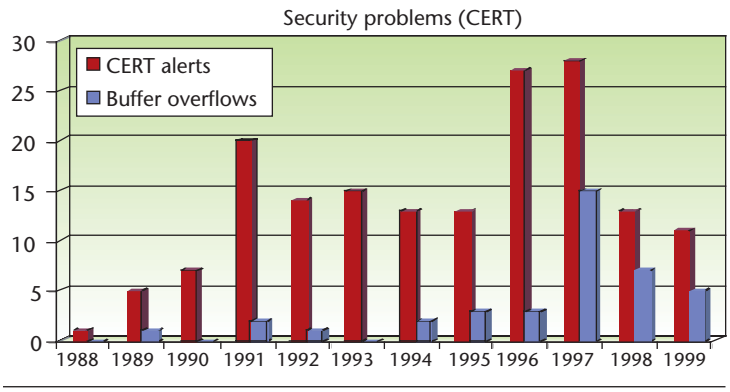

Figure 1. A study modeled after University of California, Berkeley professor David Wagner, who published a seminal paper on buffer overflow detection that included determining the prevalence of buffer overflow attacks.[5] The graph shows the number of major CERT alerts caused by the simple buffer overflow (accounting for approximately 45 percent of major security problems in this data set).

breaking systems (on purpose) in order to evaluate them. Others describe in gory detail the kinds of tools the adversary regularly wields. All of these approaches are useful.

Describing attacks

The first article in our special issue, Jonathan Pincus and Brandon Baker's "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns," deals with attacks that leverage a software security bug so common that it has become almost hackneyed—the buffer overflow. The root of this problem is one of memory management and control flow, often involving structures that make up the stack of a classic operating system. Although many popular works have examined the buffer overflow, most treatments focus on simple stack-smashing attacks.[1–3] Pincus and Baker go beyond this and demonstrate the depth and breadth of the problem.

As Figure 1 illustrates, simple buffer overflows involving stack smashing (a category of buffer overflow targeting a machine's execution stack mostly by rewriting return address fields) were very prevalent before 2000. Since then, more sophisticated buffer-overflow attacks have been dreamt up and applied in the wild. Unless system designers and defenders discuss these attacks in a deeply technical way, we run the risk of creating overly simple defenses that don't work. A canary-based dynamic defense mechanism such as Crispin Cowan's original Stackguard (which attempts to detect stack smashing by monitoring the return address just as canaries were used to protect coal miners against methane gas hundreds of years ago) might work against simple buffer overflow attacks,[4] but can be defeated with impunity by advanced approaches involving trampolining. The simple idea behind trampolining is to avoid detection by jumping over the canary/detector in multiple hops. Only by describing more sophisticated attacks in detail can we understand how well our defenses work (or don't, as the case may be). Pincus and Baker do us a great service by describing new families of buffer-overflow exploit.

Pondering policy

In Carolyn Meinel's more-political-than-technical article "Cybercrime Treaty Could Chill Research," she shares her view of current international policy and its impact on computer security. The very real threat of terrorism and its broad impact on society impels us to look at liberty in a new light. Is it true that if we outlaw guns only outlaws will have guns? Have we been trading off too much individual freedom for too little security? Where is the right balance when it comes to computer security?

Though Meinel's article reads more like an op-ed piece than a technical article, we think it's critical to consider carefully these issues. All security practitioners have a solemn duty to understand the social implications of their work and act accordingly. If we don't get involved in these issues, who will?

Advocating red teams

One common approach to system security in the practical commercial world is known as *red teaming*. The basic idea is to run attack exercises against a target system to better understand security posture and procedures. As Greg White and Art Conklin describe in "The Appropriate Use of Force-on-Force Cyberexercises," this approach is so common as to be trivial to the warfighter. We all have something to learn here.

There are many important issues to think about when assessing the value of red teams. First and most important is the idea of imposing a disciplined approach informed by knowledge and clear tactics. All too often, red teaming devolves into a feel-good exercise in which simple security problems are discovered (the so-called low-hanging fruit) and fixed; improved security is then declared even though only the most rudimentary testing has taken place. This kind of red teaming (almost undeserving of the name) is not helpful.

By analogy, consider the simple black-box testing tools application security vendors now promulgate, packaging up rudimentary sets of black-box security tests to run against a target program. These tests do provide some modicum of value, as they can demonstrate very clearly when a software security situation is a total disaster. What they can't do is provide any reasonable understanding of risk. Even if your system passes all the tests, it might still be broken. These kinds of canned tests are, in effect, badness-ometers (as opposed to security-meters) with the upper measure being "unknown." Red teaming must do more than this.

Red team exercises should be grounded in risk analysis and can be designed to raise the bar slowly and methodically over time, improving a target's security posture as they

unfold. All security testing should be designed to do this, and White and Conklin describe a reasonable approach.

Understanding the attacker's toolkit

Ironically, most of the people defending computer systems today are not programmers, but most of the people attacking our systems are. This is not good. Software practitioners must become more involved in computer security so that our collective approach is less reactive and more proactive—so we build things properly from the beginning. A good way to get started thinking about software security and software analysis is to think carefully about an attacker's toolkit.[6] At the apex of the common set of tools—including decompilers, disassemblers, fault-injection engines, kernel debuggers, payload collections, coverage tools, and flow analysis tools—is the rootkit.

The typical end to most software attacks involves the installation of a rootkit, which provides a way for attackers to return at will to machines that they now "own." Thus, rootkits, like the ones that Sandra Ring and Eric Cole discuss in "Taking a Lesson from Stealthy Rootkits," are extremely powerful. Ultimately, attackers can use them to control every aspect of a machine once the rootkit installs itself deep in the heart of a system. Of course, this idea is not exactly new. Ken Thompson of Bell Labs brilliantly exemplified the impact of rootkits and similar malicious code on software in his Turing Award–winning paper some 20 years ago.[7]

Rootkits can be run locally, or they might arrive via some other vector, such as a worm or virus. Like other types of malicious code, rootkits thrive on stealthiness. They hide away from standard system observers, employing hooks, trampolines, and patches to get their work done. They are all typically very small pieces of code and are extremely tightly written.

Sophisticated rootkits run in such a way that other programs that usually monitor machine behavior can't easily detect them. A rootkit usually provides access only to people who know that it is running and available to accept commands. The original rootkits were Trojan'ed files that had backdoors installed in them. These rootkits replaced commonly accessed executable files such as `ps` and `netstat`. Because this technique involved changing target executables' size and makeup, original rootkits could be detected in a straightforward manner using file-integrity-checking software such as Tripwire.

Today's rootkits are much more sophisticated. The kernel rootkit, for one example, is very common. They are installed as loadable modules or device drivers, and they provide hardware-level access to the machine. Because these programs are fully trusted, they can hide from any other software running on the machine. Kernel rootkits can hide files and running processes to provide a backdoor into the target machine. Understanding the ultimate attacker's tool provides an important motivator for those of us trying to defend systems.

Perhaps a science of attack is beginning to emerge. Several advanced books on attacking and breaking systems were published in 2004.[6,8,9] These books complement and extend such basic classics as the popular *Hacking Exposed*.[10] Engineers have long spent time learning from mistakes and failures; security practitioners should embrace this approach and make it their own. Readers of this special issue are getting a taste of what's to come. Computer security is becoming more sophisticated in every aspect—attacks included. Understanding the nature of attacking systems and how we can leverage attack concepts and techniques to build more secure systems is a critical undertaking. □

References

1. Aleph1, "Smashing the Stack for Fun and Profit," *Phrack*, vol. 7, no. 49, 1996; www.phrack.org/phrack/60/p60-0x06.txt.
2. M. Howard and D. LeBlanc, *Writing Secure Code*, Microsoft Press, 2002.
3. J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, 2001.
4. C. Cowan et al., "Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *Proc. 7th Usenix Security Symp.*, Usenix Assoc., 1998, pp. 63–77.
5. D. Wagner et al., "A First Step Towards Automated Detection of Buffer Over-Run Vulnerabilities," *Proc. Year 2000 Network and Distributed System Security Symp.* (NDSS'00), Internet Society, 2000.
6. G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code*, Addison-Wesley, 2004.
7. K. Thompson, "Reflections on Trusting Trust; ACM Turing Award Lecture," *Comm. ACM*, vol. 27, no. 8, 1984, pp. 761–763.
8. J. Koziol et al., *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*, John Wiley & Sons, 2004.
9. J. Whittaker and H. Thompson, *How to Break Software Security*, Addison-Wesley, 2003.
10. S. McClure, J. Scambray, and G. Kurtz, *Hacking Exposed: Network Security Secrets and Solutions*, Osborne, 1999.

Iván Arce *is chief technology officer and cofounder of Core Security Technologies, an information security company based in Boston. Previously, he worked as vice president of research and development for a computer telephony integration company and as information security consultant and software developer for various government agencies and financial and telecommunications companies. Contact him at ivan.arce@coresecurity.com.*

Gary McGraw *is chief technology officer of Cigital. His real-world experience is grounded in years of consulting with major corporations and software producers. He serves on the technical advisory boards of Authentica, Counterpane, Fortify, and Indigo. He also is coauthor of* Exploiting Software *(Addison-Wesley, 2004),* Building Secure Software *(Addison-Wesley, 2001),* Java Security *(John Wiley & Sons, 1996), and four other books. He 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.*