DIMACS Technical Report 2003-13
June 2003

From the Ground Up: The DIMACS Software Security Workshop[1]

by

Gary McGraw[2]
Cigital, gem@cigital.com

[1]Published by the IEEE Computer Society in: "On the Horizon," editors: Nancy R. Mead, nrm@sei.cmu.edu and Gary McGraw, gem@cigital.com, *IEEE Security & Privacy magazine*, March/April 2003.
[2]This material is based upon work supported by the National Science Foundation under Grant Number CCR-0302708. The author thanks NSF and also Microsoft Research for their support of the workshop on Software Security.

Computer system and network security is increasingly limited by the quality and security of the software running on constituent machines. Researchers estimate that more than half of all vulnerabilities are from buffer overruns, an embarrassingly elementary class of bugs.1 Worse, more complex problems such as race conditions and subtle design errors wait in the wings for the buffer overflow's demise. Software security problems will be with us for years, and hackers will continue to exploit systems via software defects. Clearly, a central and critical aspect of the computer security problem resides in software.

Software security-the idea of engineering software that continues to function correctly under malicious attack-is not new, but it is receiving renewed interest as reactive network-based security approaches such as firewalls prove to be ineffective. Unfortunately, today's software is riddled with both design flaws and implementation bugs, which result in unacceptable security risks.

As security researcher Steve Bellovin puts it, "any program, no matter how innocuous it seems, can harbor security holes." This notion is common knowledge, and yet developers, architects, and computer scientists only recently began systematically to study how to build secure software. In fact, the first books on software security and security engineering were published in 2001,2-4 and a body of literature on the subject has just begun to emerge. (See the "Further reading" sidebar on page 12.)

The DIMACS Software Security Workshop held in New Jersey this January (http://dimacs.rutgers.edu/Workshops/Software) explored issues such as security engineering, architecture and implementation risks, security analysis, mobile and malicious code, education and training, and open research issues. Many promising techniques have grown from connections between computer security, programming languages, and software engineering, and one workshop goal was to bring these communities closer together to crystallize the software security subfield. I report the workshop's results in this article.

A new security paradigm

Internet-enabled software applications-especially custom applications-present the most common security risks we encounter today and are the targets of choice for malicious hackers. Software security is about understanding software-induced security risks and how to manage them. Good software security practices leverages good software engineering practices: thinking about security early in the software's life cycle, knowing and understanding common threats (including language-based flaws and pitfalls), designing for security, and subjecting all software artifacts to thorough objective risk analyses and testing.

By any measure, security holes in software are common, and the problem is growing. Figure 1 shows the number of security-related software vulnerabilities reported to CERT/CC over the past several years. The evidence points to a clear and pressing need for a disciplined approach to software security.

Before we continue, I need to establish some definitions. The "Software security definitions" sidebar defines four of the most important terms we use when we discuss software security. I included them to reinforce the areas that each covers.

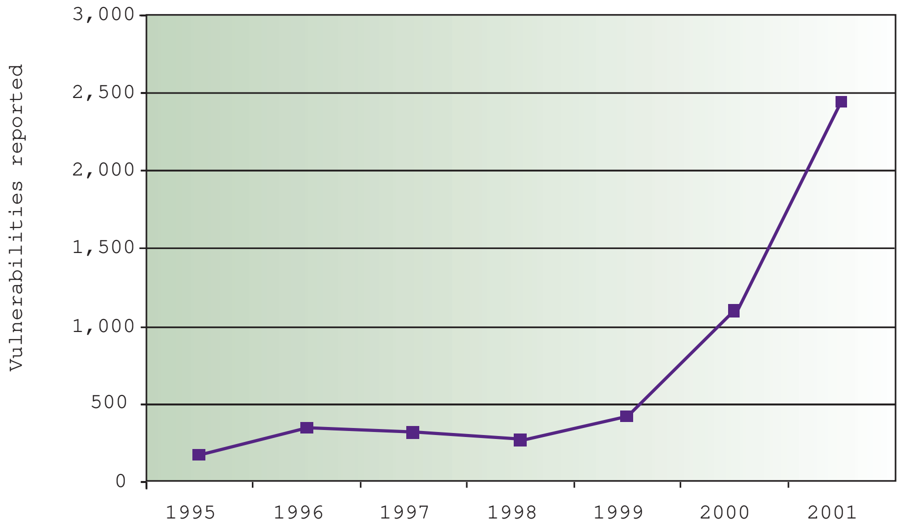

Figure 1: Security-related software vulnerabilities reported to CERT/CC over several years. This figure shows that the problem is growing, however, we don't know the exact reasons for this growth (for example, more defective code, better hacking, more code, and so on).

The trinity of trouble

Most modern computing systems are susceptible to software security problems. So, why is software security a bigger problem now than in the past? Three trends-a trinity of trouble-have heavily influenced the problem's growth and evolution. (Interestingly, these three general trends also are responsible for the rise in malicious code.5)

Reliance on networked devices

First, growing Internet computer connectivity has increased attack-vector numbers and made it easier for hackers to launch attacks. This puts software at greater risk. We are connecting ever more computers, ranging from home PCs to systems that control critical infrastructures (for example, the power grid), to enterprise networks and to the Internet. Furthermore, people, businesses, and governments increasingly depend on network-enabled communication such as email or Web pages provided by information systems. Things that used to happen offline now happen online.

Unfortunately, as these systems connect to the Internet, they become vulnerable to software-based attacks from distant sources. Attackers no longer need physical access to systems to exploit vulnerable software and shut down banking services and airlines (as shown by January's SQL Slammer worm).

Because access through a network does not require human intervention, launching automated attacks is relatively easy. The ubiquity of networking means more software systems to attack, more attacks, and greater risks from poor software security practice than in the past.

Easily extensible systems

A second trend negatively affecting software security is how extensible systems are. Extensible systems accept updates or extensions-sometimes referred to as mobile code-to evolve system functionality incrementally.6 The plug-in architecture of Web browsers, for example, makes it easy to install viewer extensions for new document types as needed. Today's operating systems support extensibility through dynamically loadable device drivers and modules, and applications such as word-processors, email clients, spreadsheets, and Web browsers support extensibility through scripting, controls, components, and applets.

From an economic standpoint, extensible systems are attractive because they provide flexible interfaces that can be adapted through new components. In today's marketplace, vendors must deploy software as rapidly as possible to gain market share. Yet the marketplace also demands that applications provide new features with each release. An extensible architecture makes it easy to satisfy both demands by letting the base application code ship early and then shipping feature extensions later, as needed.

Unfortunately, the very nature of extensible systems makes it hard to prevent software vulnerabilities from slipping in as unwanted extensions. Advanced languages and platforms including Sun Microsystem's Java and Microsoft's .NET Framework are making extensibility commonplace.

Increasingly complex systems

A third trend is the unbridled growth of modern information systems' size and complexities, especially software systems. A desktop system running Windows/XP and associated applications depends on the proper functioning of the kernel as well as the applications to ensure that vulnerabilities won't compromise the system. However, XP consists of at least 40 million lines of code, and end-user applications are becoming at least as complex.

When systems become this large, bugs cannot be avoided. Figure 2 shows how Windows' complexity (measured in lines of code) has grown over the years. The point of the graph is to emphasize the growth rate rather than the numbers. Note that the flaw rate tends to progress as the square of code size. Other factors that deeply affect complexity include whether the code is tightly integrated, the overlay of patches and other post-deployment fixes, and critical architectural issues.

The complexity problem is exacerbated by the use of unsafe programming languages such as C or C++, which do not protect against simple kinds of attacks, such as buffer overflows. In theory, we should be able to analyze and prove that a small program has no problems, but this task is impossible for even the simplest desktop systems, much less the enterprise-wide systems used by businesses or governments.

Software security definitions

The reintroduction of basic terminology with a security emphasis can help clarify things when trying to categorize problems. I propose the following usage:

- Defects. Implementation and design vulnerabilities are defects. A defect is a problem that could lie dormant in software for years and then surface in a fielded system with major consequence.
- Bug. A bug is an implementation-level software problem. Bugs might exist in code but never execute. Although many software practitioners apply the term "bug" quite generally, I reserve use of the term to encompass fairly simple implementation flaws. Bugs easily can be discovered and remedied. Researchers have made significant progress in detecting security vulnerabilities stemming from low-level and mid-level implementation bugs. Tools include FIST,1 ITS4,2 Jscan,3 Splint,4 Metal,5 and PreFix/PreFast.6 These tools detect a wide range of low-level implementation bugs including buffer overflow vulnerabilities, format string bugs, resource leaks, and simple race conditions, all of which depend on only limited code analysis and knowledge of the external environment.
- Flaw. A flaw is a problem at a deeper level. Often flaws are much more subtle than simply an off-by-one error in an array reference or use of an incorrect system call. A flaw certainly is instantiated in software code, but is also present (or absent!) at the design level. For example, a number of classic flaws exist in error-handling and recovery systems that fail in an insecure or inefficient fashion. Automated technologies to detect design-level flaws do not yet exist.
- Risk. Flaws and bugs lead to risk. Risks are not failures. Risks capture the probability that a flaw or a bug will impact the purpose of the software. Risk measures also take into account the potential damage that can occur. A very high risk is not only likely to happen, but also likely to cause great harm. Risks can be managed by technical and non-technical means.

References

[1] A. Ghosh, T. O'Connor, and G. McGraw, "An Automated Approach for Identifying Potential Vulnerabilities in Software," *Proc. IEEE Symp. on Security and Privacy*, IEEE CS Press 1998, pp. 104-114.

[2] J. Viega et al., "ITS4: A Static Vulnerability Scanner for C and C++ Code," *Proc. Ann. Computer Security Apps. Conf.*, 2000.

[3] J. Viega et al., "Statically Scanning Java Code: Finding Security Vulnerabilities," *IEEE Software*, vol. 17, no. 5, 2000, pp. 68-74.

[4] D. Evans et al., "LCLint: A Tool for Using Specifications to Check Code," *Proc. SIGSOFT Symp. Foundations of Software Eng*, ACM Press, 1994, pp. 87-96.

[5] D. Engler et al., "Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions," *Proc. Symp. OS Design & Implementation (OSDI)*, Usenix Assoc., 2000.

[6] W.R. Bush, J.D. Pincus, and D.J. Sielaff, "A Static Analyzer for Finding Dynamic Programming Errors," *Software Practice and Experience*, vol. 30, June 2000, pp. 775-802.

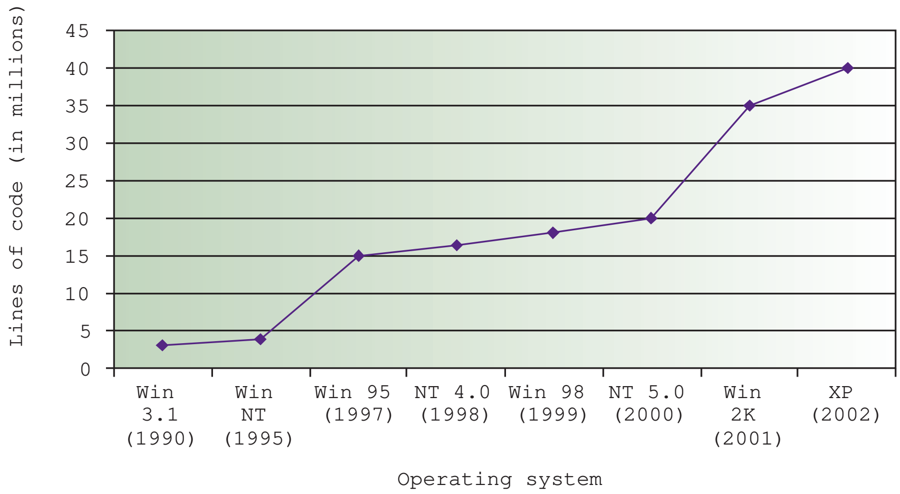

Figure 2: Growth of Microsoft's operating system code base from 1990 to 2002. These numbers reflect lines of codes for all aspects of Windows, including device drivers. The striking growth rate is apparent.

Proactive security

Software security follows naturally from software engineering, programming languages, and security engineering. It fortifies computer security by identifying and expunging problems in the software itself by building software that proactively resists attack.

For decades, computer security literature has discussed some aspects of software security but only recently has software security been recognized as a clearly defined subdiscipline. It's still in its infancy.

Architecture versus implementation

Though Figure 1 shows that software vulnerability is growing, scientists have done little work to classify and categorize software security problems.

Security vulnerabilities in software systems range from local implementation errors (for example, use of the gets() function call in C and C++), through interprocedural interface errors (for example, a race condition between an access control check and a file operation), to much higher design-level mistakes (such as error-handling and recovery systems that fail insecurely or object-sharing systems that mistakenly include transitive trust issues).

These vulnerabilities define a large range based on how much program code we must consider to understand the vulnerability, how much detail regarding the execution environment we must know to understand the vulnerability, and whether a design-level description is best for determining whether a given vulnerability is present. For example, we can determine that a call to gets() in a C and C++ program can be exploited in a buffer overflow attack without knowing anything about the rest of the code, its design, or the execution environment other than assuming that the user entering text on standard input might be malicious. Hence, we can detect a gets() vulnerability with good precision using a simple lexical analysis.

Mid-range vulnerabilities involve interactions among more than one code point. For example, precisely detecting race conditions depends on more than just analyzing an isolated line of code-it could depend on knowing about the behavior of several functions, understanding sharing among global variables, and knowledge of the operating system providing the execution environment.

Design-level vulnerabilities extend this trend further. Ascertaining whether a program has design-level vulnerabilities requires great expertise, which makes design-level flaw identification hard to do and particularly difficult to automate. Unfortunately, design-level problems appear to be prevalent and are at the very least a critical category of security risk in code. During the workshop, Microsoft reported that over 50 percent of the problems it encountered during its security push were architectural in nature.

As an example, let's consider an error-handling and recovery system. Failure recovery is an essential aspect of security engineering, but it is complicated because it interacts with failure models, redundant design, and defense against denial of service. Understanding whether an error-handling and recovery system in an object-oriented program is secure involves ascertaining a global property spread throughout many classes in typical design. Error-detection code usually is included in each object and method, and error-handling code usually is separate and distinct from the detection code. Sometimes exceptions propagate up to the system level and the machine running the code handles them (for example, Java 2 VM exception handling), which makes determining whether a given error-handling and recovery design is secure quite difficult. This problem is exacerbated in transaction-based systems commonly used in e-commerce solutions where functionality is distributed among many different components running on several servers.

Other examples of design-level problems include object-sharing and trust issues; unprotected data channels (both internal and external); incorrect or missing access control mechanisms; lack of auditing/logging or incorrect logging; ordering and timing errors (especially in multithreaded systems); and many others. To make forward progress as a scientific discipline, software security must rigorously understand and categorize problems like these.

Building secure software is like building a house. We liken correct low-level coding (such as checking array references) to using refractory bricks instead of bricks made of sawdust. The kinds of bricks that we use are important to the house's integrity, but even more important (if the goal is to keep bad things out) is having four walls and a roof in the design. The same thing goes for software: which system calls are used and how they are used is important, but overall design properties often count for more. In general, software security research has paid much more attention to bricks than to walls.

Software security areas of interest

Practitioners and scientists focus on the following areas of interest in software security:

- Reconciling security goals and software goals, software quality management in commercial practice

- Security requirements engineering
- Design for security, software architecture, and architectural analysis
- Security analysis, security testing, and the use of the Common Criteria
- Guiding principles for software security, case studies in design and analysis, and pedagogical approaches to teaching security architecture
- Software security education; educating students and commercial developers
- Auditing software; implementation risks, architectural risks, automated tools, and technology developments (such as code scanning, information flow, and so on)
- Common implementation risks: buffer overflows, race conditions, randomness, authentication systems, access control, applied cryptography, and trust management
- Application security; protecting code post production
- Survivability and penetration resistance, type safety, and dynamic policy enforcement
- Denial-of-service protection for concurrent software
- Penetrate and patch as an approach to securing software
- Code obfuscation and digital content protection
- Malicious code detection and analysis

Much work remains to be done in each of these areas, but some basic practical solutions are becoming available in the market. Examples include using simple code-scanning tools during implementation review and reactive "application firewalls."

Best practices and the state of the art

As practitioners become aware of software security's importance, they adopt and evolve a set of best practices to address the problem. Common approaches in practice today encompass training for developers, testers, and architects; analysis and auditing of software artifacts; and security engineering. Later we'll see that Microsoft has carried out one noteworthy effort under the rubric of the Trustworthy Computing Initiative.

Training and education

Software security problem awareness is growing among researchers and security practitioners. However, the most important audience has in some sense experienced the least exposure. For the most part, software architects, developers, and testers remain blithely unaware of the problem. One form of best practice involves training software development staff on critical software security issues.

The most effective form of training begins with a description of the problem and then demonstrates its impact and importance. During the Windows security push in February and March 2002, Microsoft provided basic awareness training to all its developers. Beyond awareness, more advanced software security training should cover security engineering, design principles and guidelines, implementation risks, design flaws, analysis techniques, and security testing. Quality-assurance personnel-especially those who perform testing-should have special tracks.

Today's academic curricula do little to expose students in engineering and computer science to security. Introductory programming courses can and should cover basic security tenets, because software security education is critical to changing the way software is built. One excellent technique is to have groups of students analyze the work of their peers for security problems.

Analysis, auditing, and engineering

You cannot bolt security onto an existing program or even implement particular sets of "security functionality" as a complete solution. The most generic instantiation of this problem is the common, but flawed, overreliance on cryptography. A classic rejoinder heard from software architects and developers when confronted with security is the claim, "It's secure because we use SSL." Although applied cryptography is an important weapon in the software security arsenal, it is insufficient for security. Put glibly, software security is not security software.

Because security is an emergent property of a complete system, we must take a holistic approach. We should weave in security throughout the complete software development life cycle:

- At the requirements level, we must consider security explicitly. Security requirements should not be an "add on," and should take into account emergent characteristics of security, including explicit coverage of what should be protected, from whom, and for how long.

- At the design and architecture level, a system must be coherent and present a unified security architecture, document assumptions, and identify possible attacks. At this stage of development, risk analysis is a necessity-risks can be uncovered and ranked so mitigation can begin.

- At the code level, focus on implementation flaws, especially those that can be statically discovered in an automated fashion. Note that code review, although necessary (especially when problematic languages such as C are used), is not sufficient for software security. Finding and removing all implementation flaws in source code does nothing to address architectural problems.

- When it comes to testing, security is a special case. Security testing must encompass two strategies: testing security functionality with standard functional testing techniques and risk-based testing based on attack patterns and threat models. Note that security problems are not always apparent, even when directly probing a system.

- We must monitor fielded systems during use. Simply put, attacks will happen, regardless of the strength of the design and implementation. Monitoring software behavior is an excellent defensive technique.

Successful security analysis involves two levels of system understanding: architectural risk analysis and implementation analysis. Doing only one provides a high likelihood of failure. Today, there is an over emphasis on code review, which alone cannot solve the software security problem.

Microsoft's Trustworthy Computing Initiative

Microsoft's chairman, Bill Gates, released a memo in January 2002 that highlighted the importance of building secure software to Microsoft's future. Its Trustworthy Computing Initiative has changed the way Microsoft builds software. To date, Microsoft has spent over $200 million (2,000 person days) on its software security push.

Microsoft is focusing on people, process, and technology to tackle the software security problem. On the people front, Microsoft is training every developer, tester, and program manager in basic techniques of building secure products. Microsoft enhanced its development process to make security a critical factor in design, coding, and testing of every product, for example.

There are many possible ways to integrate software security practices into the development life cycle. Figure 3 illustrates Microsoft's way by showing where "security cycles" can be applied during a standard software development life cycle.

Risk analysis, code review, security testing, and external review and testing have their places in the new process. Microsoft is building tools, including PREfix and PREfast for defect detection,7 to automate as many process steps as possible and changing its Visual C++ compiler to detect certain kinds of buffer overruns at runtime. Microsoft also has begun thinking about measurement and metrics for security.

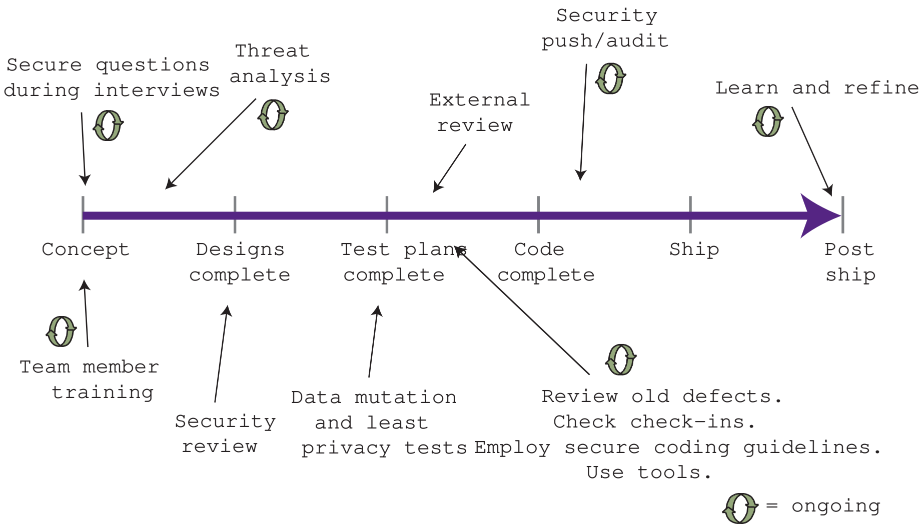

Figure 3: Microsoft has put the following software security process into place. Notice that security does not happen at one life cycle stage, nor are constituent activities "fire and forget." The idea of constantly revisiting analyses at all levels is critical to software security.

The importance of measurement

Measurement is critical to the future of software security. Only by quantifying our approach and its impact can we answer such questions as: How secure is my software? Am I better off now than I was before? Am I making an impact on the problem? How can I estimate and transfer risk?

Measurement is one foundational approach critical to any science. As Lord Kelvin put it:

> When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science.

The future belongs to the quants

We can begin to approach the measurement problem by recycling numbers from software literature. For example, we know that fixing software problems in the design stage is much cheaper than fixing them later in the life cycle. An often-quoted IBM study reports relative cost weightings as design = 1, implementation = 6.5, testing = 15, and maintenance = 100. We also know relative cost expenditures for life cycle stages: design = 15 percent, implementation = 60 percent, and testing = 25 percent. This range of numbers can provide a foundation for measuring software security impact.

Measuring ROI

A preliminary study (www.cio.com/archive/021502/security.html) reported by @stake (www.atstake.com) demonstrates the importance of concentrating security analysis effort at the design stage relative to the testing and implementation phases (see Figure 4). Separately, Microsoft reports that over 50 percent of the software security problems it finds are design flaws. We need more such analyses to advance the field.

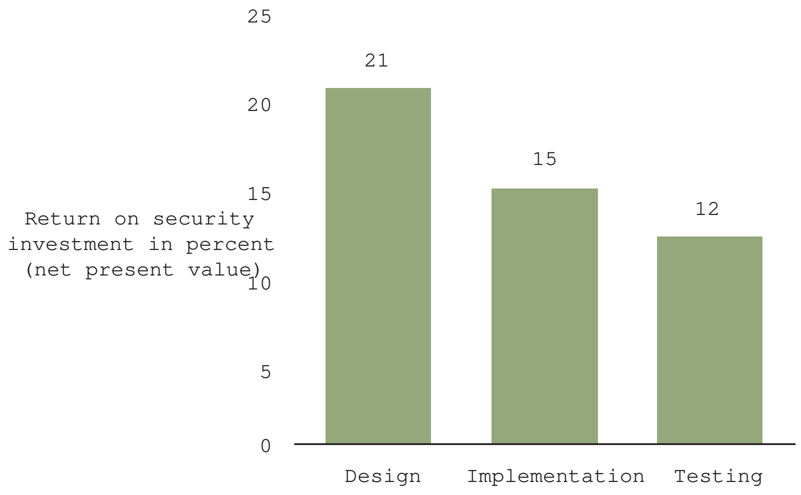

Figure 4: Phase return on investment (ROI) as measured by @stake over 23 security engagements. This study demonstrates that importance of design-level risk analysis as software security best practice.

Risk management calls out for quantitative decision support. We have work to do on measuring software security and software security risk.

A call for arms

Much work remains to be done in software security. Some of it is basic and practical (for example, making software security part of the standard software development life cycle) and some of it is far beyond current capabilities (for example, automating software architecture analyses for security flaws). The US National Science Foundation suggests using the following 11 open questions as drivers for research. There is clearly overlap among these problems, but the list raises a large number of interesting subquestions. How do we

Further reading

In addition to the explicit references included in this department, I've collected a noncomprehensive bibliography of software security publications. This list is heavily biased towards recent publications. Use these references as a springboard for more reading.

- K. Ashcraft and D. Engler, "Using Programmer-Written Compiler Extensions to Catch Security Holes," *Proc. IEEE Security and Privacy Conf.*, IEEE CS Press, 2002.
- B. Arbaugh, B. Fithen, and J. McHugh, "Windows of Vulnerability: A Case Study Analysis," *Computer*, vol. 33, Dec. 2000, pp. 52-59.
- R. Bisbey II and D. Hollingsworth, "Protection Analysis Project Final Report," tech. report ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Inst., May 1978.
- M. Bishop and M. Dilger, "Checking for Race Conditions in File Access," *Computing Systems*, vol. 9, Feb. 1996, pp. 131-152.
- C. Cowan et al., "Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *Proc. 7th Usenix Security Symp.*, Usenix Assoc., 1998, pp. 63-77.
- M. Gasser, *Building a Secure Computer System*, Van Nostrand Reinhold, 1988.
- D. Larochelle and D. Evans, "Statically Detecting Likely Buffer Overflow Vulnerabilities," *Proc. of Usenix Security Symp.*, Usenix Assoc., 2001.
- G. McGraw, "Software Assurance for Security," *Computer*, vol. 32, Apr. 1999, pp. 103-105.
- B. Miller, L. Fredricksen, and B. So, "An Empirical Study of the Reliability of Unix Utilities," *Comm. ACM*, vol. 33, Dec. 1990, pp. 32-44.
- B.P. Miller et al., "Fuzz Revisited: A Re-Examination of the Reliability of Unix Utilities and Services," tech. report CS-TR-95-1268, Univ. Wisconsin, Apr. 1995.
- F. Schneider, ed., *Trust In Cyberspace*, Nat'l. Academy Press, Washington, DC, 1998.

- Avoid building security flaws and security bugs into programs?
- Know when a system has been compromised?
- Design systems that can tolerate attack and carry out the intended mission?
- Design systems with security that can be reasonably managed?
- Provide reasonable protection of intellectual property?
- Support privacy enforcement technically?
- Get trustworthy computations from untrusted platforms?
- Prevent and withstand denial-of-service attacks?

- Quantify security trade-offs?
- Reveal and minimize assumptions in security system designs?
- Build programs and systems and know exactly what they will do and what they are doing?

We must give careful consideration to design for security. Given a set of principles and properties that we wish a system to have, we must identify design guidelines and enforcement rules. Open questions along this line of thinking include:

- Can principles be refined to guidelines?
- How can guidelines be reduced to rules that can be enforced statically?
- What technologies are suited for automated analysis?

Some concrete open research problems include explaining why the software security problem is growing; quantifying, analyzing, and explaining bug/flaw categories; performing cost/benefit analyses to prove that early is good; untangling security software from software security at the requirements stage; exploring how to teach software security most effectively to students and professionals; and inventing and applying measures and metrics.

Software security is here to stay. In the near future, awareness, training, and process integration are likely to be a big focus. A market category will emerge with impact on traditional network security shops and software development organizations. As science begins to answer the questions posed here, the field will evolve in interesting ways. Because both theoretical and practical software security aspects remain, there's plenty of work ahead. Now is the time to get in on the ground floor.

Acknowledgment

I acknowledge the thinking of all DIMACS Software Security Workshop participants, especially the program committee and the invited speakers: Ed Felten, Dan Geer, Virgil Gligor, Michael Howard, Brian Kernighan, and David Wagner. Materials from the workshop are available at www.cigital.com/ssw.

References

[1] D. Wagner et al., “A First Step Towards Automated Detection of Buffer Over-Run Vulnerabilities,” *Proc. Seventh Network and Dist. Sys. Security Symp. (NDSS)*, 2000.

[2] R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, John Wiley & Sons, 2001.

[3] J. Viega and G. McGraw, *Building Secure Software*. Addison-Wesley, 2001; *www.buildingsecuresoftware.com*.

[4] M. Howard and D. LeBlanc, *Writing Secure Code*, Microsoft Press, 2001.

[5] G. McGraw and G. Morrisett, "Attacking Malicious Code: A Report to the Infosec Research Council," *IEEE Software*, vol. 17, Sept./Oct., 2000, pp. 33-41.

[6] G. McGraw and E. Felten, *Securing Java: Getting Down to Business with Mobile Code*, John Wiley & Sons, 1999; *www.securingjava.com*.

[7] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A Static Analyzer for Finding Dynamic Programming Errors," *Software Practice and Experience*, vol. 30, June 2000.

Gary McGraw *is CTO of Cigital, where he researches software security and sets the technical vision for software quality management. He received a BA in philosophy from the University of Virginia and PhDs in cognitive science and computer science from Indiana University. He has coauthored four books: Java Security (Wiley, 1996), Securing Java (Wiley, 1999), Software Fault Injection (Wiley 1998), and Building Secure Software (Addison-Wesley, 2001) and more than 50 peer-reviewed papers. Contact him at gem@cigital.com; www.cigital.com.*