Editors: Nancy R. Mead, nrm@sei.cmu.edu
Gary McGraw, gem@cigital.com

A Call to Action

Look Beyond the Horizon

Jeannette M. Wing
Carnegie Mellon University

Today's most prevalent and widely discussed attacks exploit code-level flaws such as buffer overruns and type-invalid input. Now we should turn to tomorrow's attacks, and think beyond buffer overruns, beyond code-level bugs, and beyond the horizon. This article is a call to arms to the research community to look toward the future.

In this article, I outline a few suggestions for important research directions: software design, usability, and privacy. If we can make any progress on the first two, we could make a strong impact. I highlight the third topic because I think it deserves more attention from the scientific and technical communities, to complement the attention it already receives from the policy and legal communities. Because of my own background in software engineering, I will elaborate more on the first research direction than the other two, but I believe all three deserve equal attention.

Rationale

I make this call for three reasons: First, while we will continue to see today's code-level attacks, we can start defending against them, either by applying static and dynamic analysis tools or by coding in type-safe programming languages. We have the technical solutions in hand to detect or prevent these attacks; so it is a matter of deploying them in an effective, scalable, and practical way. I do not want to downplay the challenge in deploying technology, but I do want to distinguish between having and not having the basic science we need.

Next, security watchdog organizations such as the CERT Coordination Center (CERT/CC; www.cert.org), MITRE's Common Vulnerabilities and Exposures (CVE; http://cve.mitre.org/), and Symantec (www.symantec.com) suggest that attacks are increasingly sophisticated. As we get better at protecting our systems, the enemy gets better at attacking them. This trend likely will escalate because industry and government have highlighted security's growing importance (for example, Microsoft's Trustworthy Computing Initiative and the creation of the US Department of Homeland Security). Thus, we should be anticipating today what tomorrow's "buffer overrun" equivalent will be.

Finally, prevention is the most efficient defense. We must raise the bar on ourselves, to deploy systems that are more secure by design and more reliably implemented than those we currently know how to design and implement. We must continue to push against the limitations—technical or otherwise—of the state of the art in securing our systems. Rather than reacting to attacks, let's avoid the vulnerabilities in the first place.

Although focusing our attention on fixing today's problems is easy, the research community must look beyond the horizon. The technology we deploy to fix today's problems—for example, program analysis algorithms and strongly-typed programming languages—is based on research that started more than two decades ago. What are we doing today that will make a difference for tomorrow?

There is no silver bullet. I'm not only calling on experienced security researchers to maintain their relentless efforts, but also on researchers whose experience and expertise is not in security. Everyone needs to share the responsibility of making our systems secure; we cannot "leave it to the security guys," especially not after the fact. We all know it is better to design and build with security in mind rather than add it in as an afterthought. Also, as with much research, technological breakthroughs likely will come from those who can bring different and fresh perspectives to the table.

My highlighting the three areas of software design, usability, and privacy does not suggest that they are the only important ones. My appeal for help is to the entire research community, in all areas of computer science and related disciplines.

Girding yourself for the security research battle

Readers motivated by this call to arms to the security research community can begin to explore the three directions discussed here by reading the following references and discussing and publishing their own ideas at the conferences listed here.

Software security design

Books particularly relevant to security design and architecture:

- R. Anderson, *Security Engineering*, John Wiley & Sons, 2001.
- M. Howard and D. LeBlanc, *Writing Secure Code*, Microsoft Press, 2003.
- J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, 2001.

Also, a previous column in "On the Horizon" discussed the importance of architecture in software security:

- G. McGraw, "From the Ground Up: The DIMACS Software Security Workshop," *IEEE Security and Privacy*, vol. 1, no. 2, 2003, pp. 59–66.

Of note are the following conferences, many of which have tracks devoted to software security and architecture:

- Foundations of Software Eng., www.isr.uci.edu/FSE-12/cfp.html
- IEEE Symposium on Security and Privacy, www.ieee-security.org/TC/SP-Index.html
- International Conference on Software Engineering, http://conferences.iee.org/icse2004/
- ISOC Network and Distributed System Security Symposium, www.isoc.org/isoc/conferences/ndss/04/
- Usenix Security Symposium, www.usenix.org/events/sec04/

Usability and security

Seminal reference books on usability and security are few. Interested readers should start with more general usability books.

- J. Nielsen, *Usability Engineering*, Morgan Kaufmann, 1994
- D.A. Norman, The Design of Everyday Things, Basic Books, 2002

Usability conferences include:

- ACM Computer–Human Interaction, www.acm.org/sigs/sigchi/
- ACM Computer Supported Cooperative Work, www.acm.org/cscw2002/

Technical treatments of privacy and security

On the privacy front, several active research groups are beginning to address important technical issues. They include:

- Electronic Privacy Information Center, www.epic.org
- Privacy Forum, www.vortex.com/privacy.html

Groundbreaking technical work in privacy can be found here:

- Freedom to Tinker, www.freedom-to-tinker.com/
- Carnegie Mellon Laboratory for International Data Privacy, http://privacy.cs.cmu.edu/
- IBM Privacy Research Institute, www.research.ibm.com/privacy/
- Stanford University Privacy and Databases, http://crypto.stanford.edu/portia/

Some conferences include:

- Computers, Freedom, and Privacy, www.cfp.org/
- IEEE Symposium on Security and Privacy, www.ieee-security.org/TC/SP-Index.html

Software design and security

My first call is to the software engineering community. With security in mind, we must revisit all phases of the software life cycle: requirements, design, testing, validation, measurement, and maintenance. I will limit my remarks here primarily to software design.

Looking at software design and security simultaneously has two potential benefits. First, the security community benefits from studying attacks at a system's design and architectural levels, not just at the code level. Second, the software-engineering community benefits because coming up with generic "design" principles—for example, to evaluate when one design is better than another—often yields results that are too abstract; coming up with rules specifically for security has more likelihood of yielding results we can use in practice. Moreover, we might be able to apply variations or generalizations of these specific rules to other nonfunctional properties such as resource usage and usability.

Toward compositional security

Tomorrow's attacks will exploit vulnerabilities at software's design and architectural levels, not just at its data-structure and procedure-call levels. Design or architectural mismatches are potentially exploitable weaknesses. Often these mismatches are between a component and its operating environment, for example, because the component makes assumptions beyond what the environment can provide.

A good example of an exploitable design vulnerability is the Domain Name Service (DNS) spoofing attack.[1] The vulnerable component is the browser, which operates in the DNS infrastructure environment. To

enforce the policy that an applet must connect to the same server from which it originated, Netscape Navigator's original check used two DNS name lookups. Let *n2a* be the many-to-many relation that maps names to Internet protocol (IP) addresses, *X* be the name of the server from which the applet originated, and *Y* be the name of the server to which the applet wishes to connect. If the lookup on both names yields a nonempty intersection of IP addresses, then the assumption is that *X* and *Y* are "the same server," and we allow the connection. More succinctly,

if $n2a(X) \cap n2a(Y) \neq \emptyset$

then $\exists x \in n2a(X)\ \exists y \in n2a(Y)$ such that *connect*(x, y)

The problem is that we could establish a connection with any one of the IP addresses x in $n2a(X)$, perhaps connecting to a victim IP address in $n2a(X)$ but not corresponding to the actual originating server. The design vulnerability is that Netscape's intersection check is too weak: the existence of a nonempty intersection says nothing about machines x and y used in the actual connection; in particular, $x \in n2a(X)$ does not imply $n2a(X) \cap n2a(Y)$. The Netscape fix—storing the actual IP address i of the originating server—eliminates the first lookup and changes the intersection check to a membership check, $i \in n2a(Y)$, which, if successful, ensures that we connect to the originating server.

The point here is that the vulnerability occurred above the code level; the code correctly implemented the specified check. While the burden to patch the vulnerability was on Netscape, we also could blame the DNS infrastructure architecture: it is too easy for someone to run his or her own domain name resolver, too easy for the server n to associate false *n2a* bindings for n to arbitrary IP addresses, and too liberal, though arguably needed for flexibility, to let *n2a* be a relation rather than a many-to-one function. We also could blame Netscape for its design decision in using a weak intersection check. Or, we could blame the ambiguity of the specification itself (does "same server" mean same name or same IP address?).

A different kind of compositional attack combines several legitimate acts to produce *emergent abusive behavior*. Even a single legitimate act—multiplied many times over on the scale of the Internet—can turn into a malicious act. A prevalent simplistic example is a denial-of-service (DoS) attack. Sending a packet to a host is perfectly legitimate. A multiplicity of sends can flood the receiving host, which then shuts down, denying any further service. Moreover, multiplying this attack across a range of recipient hosts yields a distributed DoS attack. CERT/CC trends show that DoS attacks are on the rise and have surpassed buffer-overrun attacks.

Another example of emergent abusive behavior is spam: the single legitimate behavior of sending an email message multiplied many times over results in abusive behavior. A third example is making repeated queries on small data sets. A slightly more sophisticated version of this attack class is to use the reach of trusted third parties such as Google, Amazon, or eBay to gain a multiplicative factor.

These kinds of attacks are hard to define, let alone detect, because it is not clear how many is too many. Moreover, they can be subjective—what is spam to one person could be perfectly acceptable to someone else. Recovering from them can be costly, especially if the good name of trusted third parties is involved. In the extreme, they can cause people to forsake the benefits of a useful service to avoid potential annoyances. We now are at the tip of the iceberg for this kind of attack.

Here is a general framework in which to study the problem of compositional security. Let $M_1 \ldots M_n$ be n possibly different components, + be a composition operator, $\models$ be a *satisfies* relation, and ϕ be some desired security property. Ideally, we would like the following implication to hold:

$$M_1 \models \phi \wedge \ldots \wedge M_n \models \phi \Rightarrow M_1 + \ldots + M_n \models \phi, \quad (1)$$

which says that if each component M_i for $1 \leq i \leq n$ satisfies a given property ϕ, then the composition of the n components also satisfies that property. A vulnerability arises when the interfaces between any two components do not match; that is, the two components do not compose according to the meaning of composition +, for example, an assumption made by one is not discharged by the other. Emergent abusive behavior arises if n grows too large (and, presumably, ϕ captures what "too large" means). What we need to understand is what we can vary or relax in Formula 1: different notions of composition (+), different notions of satisfies ($\models$), and different kinds of properties (ϕ). For example, by fixing the first two (+ and $\models$), we can ask, "for what kind of property does the formula hold?" Or by fixing the second two ($\models$ and ϕ), we can ask, "for what relaxed notion of composition can we guarantee the given property holds in the composed system?"

This suggested framework intentionally does not fix what a component is. It can be small (for example, a procedure or class) or large (for example, a browser or database). It can be static (a class interface) or dynamic (a procedural execution). A browser and a scripting engine—each of which might be secure in its own right—when composed can lead to an entire category of attacks. In the DNS spoofing attack, the interface (or assumptions) mismatch is between the DNS infrastructure and the browser. In the spam attack, sending an email message satisfies the desired property (communication between sender and receiver); sending a multiplicity results in an undesired property (unwanted commu-

Table 1. Secure by design.

POTENTIAL PROBLEM	PROTECTION MECHANISM	DESIGN PRINCIPLES
The underlying `dll (ntdll.dll)` was not vulnerable because…	Code was made more conservative during the Security Push.	Check precondition
Even if it were vulnerable…	Internet Information Services (IIS) 6.0 is not running by default on Windows Server 2003.	Secure by default
Even if it were running…	IIS 6.0 does not have WebDAV enabled by default.	Secure by default
Even if Web-based Distributed Authoring and Versioning (WebDAV) had been enabled…	The maximum URL length in IIS 6.0 is 16 Kbytes by default (> 64 Kbytes needed for the exploit).	Tighten precondition, secure by default
Even if the buffer were large enough…	The process halts rather than executes malicious code due to buffer-overrun detection code inserted by the compiler.	Tighten postcondition, check precondition
Even if there were an exploitable buffer overrun…	It would have occurred in w3wp.exe, which is running as a network service (rather than as admininstrator).	Least privilege

(Data courtesy of David Aucsmith.)

nication between a sender and a multitude of receivers).

The challenge in achieving compositional security is that security is a global property, yet the only way we know how to build big systems is by using smaller pieces. When we put small pieces together, predicting the consequences of their composition is hard. Thus, we need to have ways that let us model, predict, and evaluate what effects putting components together have on the composed system's security.

Toward security design principles

To increase Windows Server 2003's security with respect to its predecessors, Microsoft developers abided by many design principles. The inspiration for many of those principles, such as defense in depth and principle of least privilege, comes from the security community.

To illustrate the benefits of applying security principles to software design, consider the security vulnerability reported in Microsoft Security Bulletin MS03-007 of 20 May 2003. Windows Server 2003 is unaffected by this vulnerability, but earlier versions of Windows are. The underlying vulnerability is an unchecked buffer in `ntdll.dll`, a core operating system component. One way to exploit the vulnerability is to send an ill-formed Web-based distributed authoring and versioning (WebDAV) request to an Internet Information Services (IIS) 5.0 Web server, thereby gaining control over it. (WebDAV is an extension to HTTP that lets authorized users remotely add and manage content on the Web server.) Windows Server 2003 was protected because of a series of design decisions made at different abstraction layers, as shown in Table 1.

At the innermost layer, the developers made the code more conservative by performing input validation checks. But even if they had not, at the next layer, IIS 6.0 in Windows Server 2003 (compared to IIS 5.0 in Windows 2000) does not run by default. Even if it were running by default, IIS 6.0 does not run WebDAV by default. Even it did, the ill-formed URL needed to exploit the unchecked buffer would have to be greater than 64 Kbytes and the maximum URL length IIS 6.0 allows is 16 Kbytes by default. Even if the buffer were large enough, the process would halt—rather than run the malicious code—because of buffer-overrun-detection code inserted in the compiler. Finally, even if there were an exploitable buffer overrun, the potential scope of damage would have been limited because the process would be running with network-service privileges, which are more restrictive than administration privileges.

Overall, the example illustrates the defense-in-depth principle by applying other design rules such as Secure by default and principle of least privilege at each abstraction layer.[2,3] Moreover, it also illustrates the application of security-related software design principles—for example, check precondition. Here, an implementer following robust programming practice did not assume that a precondition would hold, but checked it explicitly in case the caller had not established it.

These security design principles are well known for designing secure systems—for example, where to place firewalls and intrusion detection systems. However, we also should look beyond these principles and think of new ones specific to issues raised by software: for example, mobile code, memory management, interfacing to program libraries, synchronization constraints, race conditions, and architectural software design. We should find ways to codify them, ideally, in terms of static checks, but at least in the form of design patterns, checklists, or templates. My call to the software-engineering community is to use these principles when designing secure software.

Usability

Security is only as strong as a system's weakest link.[4] Usually, that weakest link involves the system's interaction with a human. Whether the problem is with choosing good passwords, hard-to-use user interfaces, complicated system-installation and patch-management procedures, or social-engineering attacks, the human link always will be present.

My next call to action is to the human–computer interaction community. We must design user interfaces that make security less obtrusive and less intrusive.[5,6] As computing devices become ubiquitous, we must hide security from users but still provide them control where appropriate. How much security should we have and could we make transparent to users?

We also need behavioral scientists to help computer scientists. Technologists should design systems with reduced susceptibility to social--engineering attacks. Also, as the number and nature of attackers changes in the future, we must understand attackers' psychology: from script kiddies to well-financed politically motivated adversaries. As biometrics become commonplace, we also need to understand whether and how they help or hinder security (perhaps by introducing new social-engineering attacks) or help or hinder privacy.[7]

This usability problem occurs at all system levels: at the top are users who are not computer savvy but interact with computers for work or for fun; in the middle are computer- savvy users who do not and should not have the time or interest to twiddle with settings; at the bottom are system administrators who have the unappreciated and scary task of installing the latest security patch without being able to predict the consequence of doing so.

We need to make it possible for normal humans to use our computing systems easily, but securely.

Privacy

My last call is to the general technical community. Much past privacy research addressed nontechnical questions. I believe that privacy is the next big area related to security for technologists to tackle.

There is no consensus among technologists on what privacy is, when it is violated, and so on—let alone among technologists, governments, and the general public. Computer scientists in this area must interact with policymakers, legal experts, and behavioral and social scientists to get a comprehensive scope of the issues.[8] What technical problems are possible, impossible, or impractical to solve? What must or should we leave for law and public policy to solve?

One technical viewpoint is that preserving privacy means protecting people from unauthorized information uses. Confidentiality—preventing unauthorized access to information—is thus a subcase of privacy. Technical work on privacy has focused primarily on ensuring confidentiality by analyzing information flow—for example, within a state-machine system model or among program modules.[9,10] We can annotate program variables with sensitivity labels (nonpersonal, personal, sensitive, and so on) and apply static analysis techniques to determine information leaks (for example, assigning a sensitive value to a nonpersonal variable). While this code-level work is a promising step in the right direction, I'm asking the technical community to address broader questions.

I also would like the theoretical community to design provably correct protocols that preserve privacy for some formal meaning of privacy, to devise models and logics for reasoning about privacy, to understand what is or is not impossible to achieve given a particular formal privacy model, to understand more fundamentally what the exact relationship is between privacy and security, and to understand the role of anonymity in privacy (when it is inherently needed and what the trade-off is between anonymity and forensics).

I want the software engineering community to think about software architectures and design principles for privacy. The systems community should think about privacy when designing the next network protocol, distributed database, or operating system. I would like the artificial-intelligence community to think about privacy when using machine learning to do data mining and data fusion across disparate databases. How do we prevent unauthorized reidentification of people when doing traffic and data analysis? I hope to see researchers in biometrics, embedded systems, robotics, sensor nets, ubiquitous computing, and vision address privacy concerns when designing their next-generation systems.

As a concrete goal, we need some equivalent of Butler Lampson's access matrix[11] for privacy. Once we have a formal structure that can help us think about privacy from a scientific viewpoint, we can formally define mechanisms and policies for privacy, just as we do for security. We need a characterization of the direct and hidden relations among users, their data, their control over data, their control over subsequent release and use of their data, and how these relations change over time. Another concrete goal would be to relate policy with technology, I would like to see how we could codify privacy policies of "fair information practices" (notice, choice, access, security, and redress) and check them in software.

Privacy is getting a lot of attention in the press because of homeland security, computer-assisted passenger prescreening, radio-frequency identification tags, identity theft, and so on. It will become even more important as computing becomes more ingrained in our daily lives. Privacy would be at the heart of our democratic society if society could trust electronic voting. Thus, it is a timely opportunity for scientists to step up to the technical challenges privacy raises.

I view the security problem as a race between the good guys and the bad guys. Usually, the good guys are trying to catch up or just stay even. The security problem is not going to go away anytime soon; it has been with us since the computing age began, and no matter how much money we throw at it, it won't completely disappear. The problem is on our radar screens today because of increased security demand by businesses and increased awareness by end users (usually in an unfortunate way—by having to install a critical update for the latest security vulnerability).

My call to action is to the good guys—to look beyond the horizon. While we continue to slog through today's buffer-overrun problems, we need to stay even with the bad guys, who are ready to attack at system and architectural levels. Working together, keeping our focus on and beyond the horizon, will ensure we do not fall too far behind in the security race. □

Acknowledgments

Much of my thinking presented in this article was done during a one-year sabbatical at Microsoft Research in Redmond, Washington. I thank Jim Larus, Amitabh Srivastava, Dan Ling, and Rick Rashid for hosting my visit. I also thank all the attendees of the UW-MSR-CMU Software Security Summer Institute (http://research.microsoft.com/projects/SWSecInstitute/) for their lively discussions, which helped sharpen my thoughts. I heard many of the ideas and examples presented in this article from some of these participants. In particular, I thank Tom Longstaff for his statistics pointing out the growing DoS trend; Udi Manber for introducing me to the term emergent abusive behavior*; Steve Lipner for presenting Mike Howard's slides, which include Dave Aucsmith's MS03-007 example; and Doug Tygar for his talk about privacy architectures. Finally, I thank Jon Pincus, my close working colleague at Microsoft, for our endless conversations and technical discussions during my sabbatical. Many of Jon's ideas, especially on privacy, are reflected herein.*

References

1. D. Dean, E.W. Felten, and D.S. Wallach, "Java Security: From HotJava to Netscape and Beyond," *Proc. 1996 IEEE Symp. Security and Privacy*," 1996, IEEE Press, pp. 190–200.
2. J. Saltzer and M. Schroeder, "The Protection of Information in Computer Systems," *Proc. IEEE*, vol. 63, no. 9, 1975, pp. 1278–1308.
3. J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, 2001.
4. B. Schneier, *Secrets and Lies: Digital Security in a Networked World*, John Wiley & Sons, 2000.
5. M.A. Sasse, S. Brostoff, and D. Weirich, "Transforming the 'weakest link'—A Human/Computer Interaction Approach to Usable and Effective Security, *BT Tech. J.*, vol. 19, no. 3, 2001, pp. 122–131.
6. A. Whitten and D. Tygar, "Why Johnny Can't Encrypt," *Proc. 8th Usenix Security Symp.*, Usenix, 1999, pp. 169–184.
7. L. Palen and P. Dourish, "Unpacking Privacy for a Networked World," *Proc. Conf. Human Factors in Computing Systems*, ACM Press, 2003, pp. 129–136.
8. E.W. Felten, "Freedom to Tinker;" www.freedom-to-tinker.com/archives.
9. J.A. Goguen and J. Meseguer, "Security Policies and Security Models," *Proc. 1982 IEEE Symp. Security and Privacy*, IEEE Press, 1982, pp. 11–20.
10. A. Sabelfeld and Andrew Myers, "Language-Based Information-Flow Security," *IEEE J. Selected Areas in Comm.*, vol. 21, no. 1, 2003, pp. 5–19.
11. B. Lampson, "Protection," *Operating Systems Rev.*, vol. 8, no. 1, 1974, pp. 18–24.

Jeannette M. Wing *is professor of computer science, associate department head for the PhD program, and associate dean for academic affairs at Carnegie Mellon University. Her research interests include software specification, verification, and security and programming languages and methodology. She has a PhD in computer science from MIT. She is an IEEE and an ACM fellow. Contact her at wing@cs.cmu.edu.*