Beyond the Horizon: A Call to Arms

Jeannette M. Wing
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

October 3, 2003

1 Introduction

This article is a call to arms to the research community. Today's attacks exploit code-level flaws such as buffer overruns and type invalid input. I would like the attention of the research community to turn to tomorrow's attacks—to think beyond buffer overruns, beyond the level of code, beyond the horizon.

There are three reasons I make this call. First, while we will continue in the future to see today's kinds of code-level attacks, in principle we have the technology for fending them, either by applying static and dynamic analysis tools or by coding in type-safe programming languages. Thus, we have the technical solutions in hand to detect or prevent these attacks; it is a "mere" matter of deploying them—in an effective, scalable, and practical way.[1]

Second, the trends monitored by security watchdog organizations such as SEI/CERT, MITRE/CVE, and Symantec suggest that attacks are getting more sophisticated. As we get better at protecting our systems the enemy gets better at attacking them. This trend will likely escalate since industry and government both have highlighted the growing importance of security (e.g., Microsoft's Trustworthy Computing Initiative and the creation of the Department of Homeland Defense). Thus, we should be anticipating today what the buffer overrun of tomorrow will be.

Finally, prevention is the most efficient defense. It eliminates classes of attack from the get-go. We need to raise the bar in our own efforts to deploy systems that are more secure by design and more reliably implemented than those we know how to design and implement today. We need to continue to push against the limitations, be they technical or not, of the state of the art in securing our systems. Rather than be attacked and then react, let's avoid the vulnerabilities in the first place.

While it is easy to focus our attention on fixing today's problems, it is up to the research community to look beyond the horizon. The technology we are deploying for fixing problems now, e.g., program analysis algorithms and strongly-typed programming languages, is based

[1]This matter is not to downplay the challenge in deploying technology. I simply want to distinguish between having the basic science needed and not having it.

on research that started over two decades ago. What are we doing today that will make a difference for tomorrow?

There is no silver bullet. My call to arms is not just to experienced security researchers to keep up their relentless efforts, but also to researchers whose experience and expertise is not in security. I imagine sprinkling researchers in all areas of computer science and related disciplines with some "security fairy dust." We all need to share the responsibility of making our systems secure; we cannot "leave it to the security guys," especially not after the fact; we know it is better to design and build with security in mind rather than to add it in as an afterthought. Also, as with much research, technological breakthroughs will likely come from those who can bring different and fresh perspectives to the table.

Below I outline a few suggestions for important research directions: software design, usability, and privacy. For the first two, if we can make any progress, we have the potential of having a high impact. I highlight the third research direction 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 on the first research direction more than the other two, but I believe all three deserve equal attention.

By no means should my highlighting these three areas 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.

2 Software Design and Security

My first call to arms is to the software engineering community. We need to revisit all phases of the software lifecyle with security in mind: requirements, design, testing, validation, measurement, and maintenence. I will limit my remarks here primarily to software design.

Looking at software design and security together has two potential benefits. First, the security community will benefit from studying attacks at the design and architectural levels of a system, not just the code level. Second, the software engineering community will benefit since coming up with generic "design" principles (e.g., to evaluate when one design is better than another) often yields results that are too abstract; however, coming up with rules specifically for security has more likelihood of yielding results that can be operationalized, and hence used in practice. Moreover, we may be able to apply variations or generalizations of these specific rules to other non-functional properties.

2.1 Toward Compositional Security

Tomorrow's attacks will exploit vulnerabilities at the design and architectural levels of software, not just the data structure and procedure call level. Design or architectural mismatches are vulnerabilities that are potential weaknesses to exploit. Often these mismatches are between a component and its operating environment, e.g., because the component makes assumptions that are stronger than what the environment can uphold.

An old, but canonical example of an exploitable design vulnerability is the Domain Name Service spoofing attack. The system component in question is the browser, which operates in the environment of the DNS infrastructue. To enforce the policy that an applet connect to the same server from which it originated the Netscape browser's original check used two DNS lookups on names. Let *n2a* be the many-to-many relation that maps names to 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 establish a connection with any one of the ip addresses x in $n2a(X)$; thus, we could very well be connecting to a victim ip address that is in $n2a(X)$ but does not correspond to the actual originating server. The design vulnerability is that Netscape's intersection check is too weak: the existence of a non-empty intersection says nothing about the machines x and y used in the actual connection; in particular, $x \in n2a(X)$ does not imply $x \in n2a(X) \cap n2a(Y)$. The Netscape fix, by 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 is that the vulnerability occurs above the level of code: the check was correctly implemented as specified. While the burden to patch the vulnerability was on Netscape, we could also blame the architecture of the DNS infrastructure: it is too easy for someone to run his or her own domain name resolver; it is too easy for the server with name n to associate false *n2a* bindings for n to arbitrary ip addresses; it is too liberal, though arguably needed for flexibility, to have *n2a* be a relation rather than a many-to-one function. We could also blame Netscape for its design decision in using an erroneous weak intersection check. Or, we could blame the ambiguity of the specification itself (what does "same server" mean: same name or same ip address?).

A different kind of compositional attack is a collection several legitimate acts which when combined together results in *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 attack. Sending a packet to a host is perfectly legitimate; it is the basis of communication. A multiplicity of sends can result in flooding the receiving host, which then shuts down, denying any further service. Moreover, multiplying this attack across a range of recipient hosts yields a simple distributed denial-of-service attack. Trends reported by CERT show that denial-of-service attacks are on the rise and already have overtaken buffer overrun attacks in number. Another simple 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 class of attack is to use the reach of trusted third parties such as Google, Amazon, or ebay to 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 may be perfectly acceptable to someone else. They can be costly to recover from, 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 should expect to see more and more of this kind of attack in the future; we are at the tip of the iceberg now.

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:

$$M_1 \models \wedge\phi \ldots \wedge M_n \models \phi \Rightarrow M_1 + \ldots + M_n \models \phi$$

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 if the interfaces between any two components do not match; that is, the two components do not compose according to the meaning of composition (+), e.g., 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 are what we can vary or relax in the formula above: different notions of composition (+), different notions of satisfies ($\models$), and different kinds of properties (ϕ). For example, fixing the first two (+ and $\models$), we can ask for what kind of property does the formula hold? Or 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, e.g, a procedure or class, or more relevantly, large, e.g., a browser or database. It can be static, e.g., a class interface, or dynamic, e.g., the execution of a procedure.

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 to put smaller pieces together. When we put small pieces together it is hard to predict the consequences of their composition. Thus, we need to have ways that allow us to model, predict, and evaluate what effect putting components together has on the security of the composed system.

2.2 Toward Security Design Principles

To increase the relative security of Windows Server 2003 with respect to its predecessors, Microsoft developers abided by many design principles. The inspiration for many of these 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 the Microsoft Security Bulletin MS03-007. Windows

Server 2003 is unaffected by this vulnerability, whereas earlier versions of Windows are. The underlying vulnerability is due to an unchecked buffer in a core operating system component, `ntddl.dll`. One way to exploit the vulnerability is to send an ill-formed WebDAV request to a IIS 5.0 web server and thereby gain control over the web server.[2] Windows Server 2003 was protected because of a series of design decisions made at different abstraction layers, as shown in Figure 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 (as opposed to IIS 5.0 in Windows 2000) does not run by default. But even if it were running by default, IIS 6.0 does not run WebDAV by default. But even it did, the ill-formed URL needed to exploit the unchecked buffer would have to be greater than 64 KB and the maximum URL length allowed by IIS 6.0 is 16 KB by default. But even if the buffer were large enough, the process would halt, rather than run the malicious code, because developers compiled their source code with the -GS switch, which has the effect of inserting compile-time code to make it difficult to exploit unchecked input (e.g., making it difficult to alter the return address). Finally, even if there were an exploitable buffer overrun, the potential scope of damage would have been limited since the process would be running with only "network service" privileges, which are more restrictive than "admin" privileges. Overall, the example illustrates the Principle of Defense in Depth, by applying at each abstraction layer other design rules such Secure by Default and Principle of Least Privilege. Moreover, it also nicely illustrates the application of software design principles, e.g., Check Precondition, in the security context; here, an implementer following robust programming practice will not assume a precondition holds, but check it explicitly in case the caller had not established it.

These security design principles are well-known for designing secure systems, e.g., where to place firewalls and intrusion detection systems. My call to the software engineering community is to revisit these principles in the design of secure software.

3 Usability

Security is only as strong as the system's weakest link. More often than not that weakest link involves the system's interaction with a human being. 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 will always be present.

My next call to arms is to the human-computer interaction community. We need to design user interfaces to make security both less obtrusive to and less intrusive on the user. As computing devices become ubiquitous, we need to hide security from the user but still provide user control where appropriate. How much of security should we and can we make transparent to the user?

[2]WebDAV is a distributed author and versioning protocol extension to http, allowing authorized users to add and manage content on the web server remotely.

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 ...	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 WebDAV had been enabled ...	The maximum URL length in IIS 6.0 is 16KB by default ($>$ 64 KB 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 "network service" (rather than "admin").	Least Privilege

Figure 1: Secure by Design: Windows Server 2003 Unaffected by MS03-007. Example from David Aucsmith.

We also need behavioral scientists to help the computer scientists. Technologists need to design systems to reduce their susceptibility to social engineering attacks. Also, as the number and nature of attackers change in the future, we need to understand the psychology of the attacker: from script kiddies to well-financed, politically motivated adversaries. As biometrics become commonplace, we need to understand whether and how they help or hinder security (perhaps by introducing new social engineering attacks). Similarly, help or hinder privacy?

This usability problem occurs at all levels of the system: at the top, users who are not computer savvy but interact with computers for work or for fun; in the middle, users who are computer savvy but do not and should not have the time or interest to twiddle with settings; at the bottom, 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 human beings to use our computing systems easily, but securely.

4 Privacy

My last call to arms is to the technical community in general. Much past research in privacy addresses non-technical 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, etc., let alone among technologists, governments, and the general public. Computer scientists in this area will need to interact with policymakers, legal experts, and behavioral and social scientists to get a comprehensive scope of the issues. As scientists, we need to be able to answer: 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 uses of information. Confidentiality, defined as unauthorized access to information, is thus just a subcase of privacy. Technical work on privacy has focused primarily on ensuring confidentiality by analyzing information flow, e.g., within a state machine model of a system or among modules in a program. For example, we can annotate program variables with sensitivity labels (e.g., non-personal, personal, and sensitive) and apply static analysis techniques to determine information leaks (e.g., assigning a sensitive value to a non-personal variable).

While this code level work is a promising step in the right direction I again would like to raise grander questions for the broader technical community to address. I 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 model of privacy, to understand more fundamentally what the exact relationship is between privacy and security, and to understand the role of anonymity in privacy (when is it inherently needed and what is the tradeoff between anonymity and forensics). I would like the software engineering community to think about software architectures and design principles for privacy. I would like the systems community to 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 disjoint databases. How do we prevent unauthorized reidentification of people when doing traffic and data analysis? I would like to see researchers in biometrics, embedded systems, robotics, sensor nets, ubiquitous computing, and vision address privacy concerns along with the design of their next-generation systems.

As a concrete goal, I would like to see some equivalent of Lampson's access matrix for privacy. Once we have a formal structure that can help us think about privacy from a scientific viewpoint, then we can design mechanisms and policies for privacy, just as we do for security (narrowly defined). We need a characterization of the direct and hidden relations among users, their data, their control over data, and their control over subsequent release and subsequent use of their data. Moreover, these relations change over time.

As another concrete goal to relate policy with technology, I would like to see how privacy policies of "fair information practices" (notice, choice, access, security, redress) can be codified or checked in software.

Privacy is getting a lot of attention in the press because of Terrorism Information Awareness (TIA), Computer Assisted Passenger Prescreening System (CAPPS II), Radio Frequency Identification (RFID) tags, identity theft, etc.. It will become even more important as computing becomes more ingrained in our daily lives. It will be at the heart of our democratic society, if electronic voting is to be trusted by society. Thus, it is a timely opportunity for scientists to step up to the technical challenges privacy raises.

5 Closing Remarks

I view the security problem as a race—between the good guys and the bad guys. The good guys are almost always trying to catch up or stay even with the bad guys. The security problem is not going to go away anytime soon; it has been with us since Day One of the computing age and no amount of money thrown at it will make it completely disappear. The problem is on our radar screen today because of increased demand for security 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 arms is to the good guys—to look beyond the horizon. While we continue to slog through the buffer overrun problems of today, we need to be apace with the bad guys who are all ready to operate at vulnerability levels higher than the code.

My call to arms is to the broad research community, in all areas of computer science as well as related areas that are affected by computing or that can inform the technologists. Working together, keeping our focus on and beyond the horizon, can help ensure we do not fall too far behind in the security race.

Acknowledgments

Much of my thinking presented in this article was done while I spent a one-year sabbatical at Microsoft Research in Redmond, WA. I would like to thank Jim Larus, Amitabh Srivastava, Dan Ling, and Rick Rashid for hosting my visit. I would also like to thank all the attendees of the UW-MSR-CMU Software Security Summer Institute [3] 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 his introducing me to the term *emergent abusive behavior*; Steve Lipner for presenting Mike Howard's slides, which includes Dave Aucsmith's MS03-007 example; and Doug Tygar for his talk about privacy architectures. Finally, I would like to 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.

[3]`http://research.microsoft.com/projects/SWSecInstitute/index.htm`