Editors: Marty Stytz, mstytz@att.net
James Whittaker, jw@cs.fit.edu

Why Secure Applications Are Difficult to Write

James Whittaker
Florida Tech

Developing good software is hard enough, but when software must be both good and secure, the result is many extra layers of complexity that, unfortunately, the computer science community is only just learning how to handle. In this issue's installment, we discuss software security as an awareness issue, because software that is unaware of its potential vulnerabilities has no chance of being secure. Only by being aware of potential threats and using vigilant defensive countermeasures can we hope to write secure applications, and we can only achieve security through awareness and understanding of these issues.

I categorize application security concerns into four specific issues: input streams, output production, internal data, and algorithms and computation.

The first two concerns—input and output—are related to the environment in which applications execute. The last two—data and algorithms—are related to an application's awareness of its own internal secrets. Those secrets could be the data an application stores or the algorithm it uses to perform its work. All four issues relate to awareness: secure software must always be aware of what is going on, both inside its perimeter and out to respond effectively to malicious threats.

Securing input

Inputs are the mechanism that software employs to communicate with its users. In the late 1980s and early 1990s, the software reliability engineering community, particularly the Cleanroom community, pushed tests prioritized by their relationship to actual use; tests that users are likely to apply were given the highest priority. But hackers are users too, and they are much more likely to apply input combinations that normal users would never think of. Security testing must, therefore, take a different focus than reliability testing to account for malicious users.

Malicious input can often cause software to fail (resulting in denial of service) or to execute foreign instructions (letting a remote user execute code on another person's computer). Inputs must be carefully checked for validity before they are processed, or the application could cause undesirable and insecure behavior.

Securing input interfaces requires a series of checks on inputs and their sources to ensure that users are who they say they are and that their input will not compromise the system on which the application resides.

The first line of defense is user authentication—software must identify a user as a valid entity with which it is authorized to communicate. Malicious users who gain access through or around authentication code are particularly dangerous because the application will trust them. Techniques for authenticating users range from challenge–response password entry to biometric devices that read hand geometry, fingerprints, or retinal patterns. The idea of communication with only legitimate, friendly users is appealing, but authentication schemes are imperfect and subject to counterfeiting and forgery. Therefore, a more complete authentication scheme is often required.

A second line of defense is guarding against authenticated users applying inputs that would cause their privilege level to escalate or let them access restricted functionality. This means that even trusted users must be monitored and any unauthorized behaviors must be stopped. Good security is not achieved through perimeter defense; it requires knowing who a user is as well as what that user is doing.

A third line of defense comes with the realization that input validation and user authentication are not tasks that are performed only once. A malicious user might try to assume a legitimate user's identity after the authentication process has succeeded; alternatively, they might try to add malicious content to a data file after the file has been validated. These "bait and switch" attacks are common and require software to be constantly vigilant against users and inputs that are not what they seem to be.

Further reading

Several books on secure software development have surfaced recently. Michael Howard and David LeBlanc's excellent book *Writing Secure Code* (Microsoft Press, 2001) should be required reading for every developer. It is a good resource of Windows-specific security issues that developers must navigate and contains a vast store of secure coding wisdom that any programmer would be foolish to ignore. Gary McGraw and John Viega's *Building Secure Software* (Addison-Wesley, 2001) is also a treasure trove of warnings, advice, and techniques that educates developers on a wide array of security concerns.

The book that remains the classic in the cryptology field is Bruce Schneier's *Applied Cryptography* (Wiley, 1994), but my favorite, general-purpose security book is Ross Anderson's *Security Engineering* (Wiley, 2002). It explains the field from top to bottom in a readable and understandable fashion. If every computer science student read these two books, the digital world would be a much more secure place. Finally, I offer my book *How to Break Software* (Addison-Wesley, 2002) as the closest thing the world has to a testing book for software security. Although it focuses on more general functional concerns, it shows how software can be broken and thus gives developers insights into the minds of their adversaries. Happy reading.

In the event that inputs come from inherently untrustworthy sources such as the Internet, each attribute of such inputs must be validated. To ensure security, inputs must be the correct type, of acceptable length (to avoid buffer overflow), and have content that the application can securely process. Executable content must be used cautiously.

Secure applications should be paranoid about with whom they are communicating and what information they convey through that communication. Applications that process security-critical data must be vigilant against both users (whether they are trusted or not) and the input that they supply.

Securing output

An application's secrets are often delivered to intended recipients by transmitting a file or displaying output on a monitor. Authenticating the output's recipients is just as difficult as authenticating users who supply input—and just as important.

For secrets that are transmitted over a network, it is crucial that we use strong, well-implemented encryption, which could well be the most studied aspect of modern computer security. Indeed, many strong ciphers are impossible to crack in a reasonable time; however, shoddy implementations of strong ciphers are readily cracked. If an encryption key is inadvertently stored in a conspicuous place (such as the system registry or a file), then the cipher's strength has little effect. Likewise, the cipher will be weak and the information likely exposed if poor, random number-selection algorithms encrypt it.

Secret information should always be encrypted when transmitted over a network or at rest on a file system. Copy protection might also be necessary for certain data types. However, the information eventually must be decrypted and displayed for user consumption. This is when protection is at its most difficult—because information stored in memory can be available to other processes and is subject to being copied or even captured via something as innocuous as a screen dump.

Securing outputs so that they are available to only legitimate users is perhaps even more difficult than securing inputs. Application developers must be aware of and constantly vigilant against potential threats. Without awareness of the ways and means that an attacker can use to access an application's secrets, there is little chance that those secrets will be adequately protected.

Securing internal data

Securing internal data is much the same as securing external outputs. Sensitive data should be encrypted when at rest and validated when used. Data in use should be minimally exposed to potential attackers.

The biggest exposure point is data in memory. All data must eventually hit a memory location at some time; if the data is used in some computation or sent to an output device, then it must be stored in memory unencrypted. This is when it is most vulnerable. Techniques to obfuscate memory usage, flush memory, and lock interprocess memory access are an application's best defense against exploitation of its secret data.

Developers must be aware of when, and for how long, sensitive data remains in the memory's unprotected state. They must also be aware of the ways in which they can protect such data and the protection mechanisms' technical limitations.

Securing algorithms and computation

There are two aspects to securing an application's internal code. The first is identical to securing data; if an algorithm is proprietary and perceived as at risk of being reverse engineered, then it should be encrypted when at rest and protected from debugging while being executed. This includes simple, operating-system-provided antidebugging APIs and more complicated obfuscation and memory-protection mechanisms.

The second aspect of code security is preventing code from performing insecure behavior. The first rule of secure coding should be that of *least privilege*; that is, the code should run with the minimum amount of privilege necessary to perform its prescribed tasks. Code should run with admin or root privilege only if it is absolutely necessary to do so. And if such privilege is necessary, then additional security precautions are definitely warranted.

If an application has code that monitors network traffic for input, such as a Web or mail server, then it should open only those ports that have traffic with which it is interested. The less contact an application has with raw network traffic, the less likely it is to be tricked into performing ill-advised computation.

Not all security problems can be solved with software. Software pirates can still steal a vendor's proprietary music with old-fashioned recording devices. Hackers can use cameras to photograph sensitive information on a computer screen. But awareness of the categories of problems that do exist and that can be solved with software is a start. In future columns, we will be exploring potential problems and solutions in greater detail. □

James Whittaker *is a professor of computer science at Florida Tech and director of the Center for Information Assurance. His research interests are tools and methods for penetration testing, reverse engineering, binary instrumentation, and software protection. He received a BA in computer science and mathematics from Bellarmine College, and an MSc and PhD in computer science from the University of Tennessee. He is the author of* How to Break Software *(Addison-Wesley, 2002). He is a member of the ACM and IEEE. Contact him at jw@cs.fit.edu.*