Teaching Secure Programming

Computer users, managers, and developers agree that we need software and systems that are "more secure." Such efforts will require support from both the education and training communities to improve software assurance—particularly in writing secure code.

Matt Bishop
University of California, Davis

Deborah A. Frincke
Pacific Northwest National Laboratory

Discussions of what should be taught inevitably circle back to who is teaching, who is learning, and what the ultimate purpose should be. Key differences exist between curricula designed to enhance an individual's skill set (the specific ability to write secure code, for example) and curricula designed to enhance an individual's knowledge (the principles behind writing secure code). Both types are crucial to lasting improvement in the state of modern software products. If we develop and polish only curricula intended to enhance specific individual skill sets, the rapid pace of technological change will soon render the skills obsolete, and the individuals involved will require constant retraining. If we develop only knowledge, we'll have many people who know what principles should be applied but who might not be able to produce good secure code.

What is assurance?

"Assurance" has nearly as many definitions as definers. In this article, we consider it to be a measure of confidence that some entity—perhaps a computer program—meets its requirements. As a parallel, "safety assurance" for a bridge would encompass demonstrable measures that increase our confidence that the bridge is safe to use in its intended environment. "Security assurance" emphasizes requirements involving security. A practitioner who demonstrably "writes secure code" is illustrating a skill that supports increased confidence that a system using that code will be secure. As with the bridge, this view of security assurance is tied to an understanding of the environment of use, the anticipated uses to which the resultant system or code will be put, and some definition of what "safe" (or secure) means in this context. Divergence from any of these expectations moves the system outside the known area and could result in failure, just as driving a too-heavy truck over a bridge could result in collapse.

In higher education, the primary goal of teaching assurance is not to teach students how to avoid buffer overflows in a particular programming language, but rather to teach them about the principles to use in determining which programming practices to follow to prevent the issue of buffer overflows from arising. The difference between these approaches is the difference between higher education and training.

Designing security in

The emphasis in higher education is on designing security in from the beginning rather than adding it afterward. Adding security mechanisms to, or repairing vulnerabilities in, an existing system can create several problems, and might not even fix the one being repaired. The new mechanism must be designed to work correctly with all aspects of the existing system, software, and environment—a great challenge, considering the potential for errors or misunderstandings.

Security patches can introduce new security vulnerabilities, as one vendor discovered upon fixing a widely publicized security flaw that let attackers read users' email: the patch introduced a new flaw that had the same effect as the old one—and was easier to exploit.

Mismatches between new security mechanisms and existing software cause interface problems, which, in turn, cause security problems. In the best-known example, one set of software assumed the inputs were in metric units, whereas the other assumed English units; the result was the loss of a spacecraft.[1]

The added mechanism itself can also create security problems. For example, consider the Windows XP Service Pack 2, which appreciably strengthened the security of Windows XP. Unfortunately, it also interfered with certain existing programs, preventing some from working—a denial of service.[2] The service pack strengthened the systems of users who didn't run those programs, but it created a new security problem for those who did.

Generality

By avoiding a focus on particular security vulnerabilities, we can teach

students to identify potential problems by analyzing the means used to implement systems and software. Rather than focusing on arrays that overflow, for example, a student would learn that variables implicitly impose size constraints—an integer, for instance, must be between 2,147,483,647 and –2,147,483,648, inclusive. The appropriate question to ask is what happens when this assumption is violated. If we consider array types as including their sizes—for example, an array of 100 characters—the same principle applies, and the buffer-overflow problem is an aspect of a more general, and pervasive, problem. Learning to handle buffer overflows won't help a programmer handle numeric overflows, but learning how to deal with overflows in general lets the programmer handle both types. The more general approach is the hallmark of higher education.

This emphasis on generality lets students apply concepts and principles to any programming language and environment. Take programming languages, for example. The type-checking features in languages such as Modula 2 and Python reduce the chances of calling modules incorrectly. Contrast them with those of the C programming language, or assembly, in which the wide use of mixing different types often leads to problems. Security features (such as bounds-checking for buffers) are built into languages such as Java, whereas C++ and others have no such features. Hence, the classes of security problems that can arise in programs vary, in part, according to the language. Teaching specifics of security-enhanced programming for a particular set of languages helps only when a student is working with those languages. The student can work only by analogy when dealing with other languages, and if they allow classes of flaws that the other languages don't, the student might not be able to write programs that anticipate that set of problems. If students understand the more general problems, they can specialize with particular programming languages.

Principles-based approach

In the late 1960s, scientists began worrying about the security implications of storing information on computers. Although discussed only in the context of classified information, the problems they recognized included confidentiality, integrity, and availability. The Ware report, published by the RAND Corporation, identified computer security as an area of concern.[3] The Anderson report, commissioned by the US Air Force, suggested approaches for solving some problems, and identified some other specific threats, including the trojan horse.[4] Although discussing mainframe systems, these reports presented universal concepts and principles. Much modern work simply takes these ideas and applies them to current systems.

In academia, these principles are central to teaching assurance. Consider the concept of a *reference-validation mechanism*, embodied in a *reference monitor*, which is a system component that controls access to a given resource. The reference monitor must meet three requirements:

- all access to the resource must go through it;
- it must be simple, so that it can be verified; and
- it must be tamperproof, so that it can't be modified illicitly.

Now apply this concept to a simple Web server, which is designed to send files (Web pages) to clients. These files must be inaccessible to all other network servers. This is a configuration issue that requires us to address some programming considerations (such as permissions checking) and management decisions (where to put the files). The simplicity requirement means that the server should be as small as possible, with functionality only to serve the Web pages. It should not, for example, execute a subprocess to read the files. Finally, the requirement that the program be tamperproof means that the server should prevent any input from altering the executing code. This requires the prevention of buffer overflows (which typically inject executable code into the process), among other problems. As we add requirements to the Web server—for example, the ability to execute a specific set of common gateway interface (CGI) programs—the application of these principles changes (the tamperproof requirement now extends to the CGI programs executing on behalf of the Web servers). Nonetheless, the principles remain the same.

This explains why it's important to study the history of assurance. The early papers present seminal work that's still in use today, although in different environments. Santayana's maxim is true here: those who do not know history are doomed to repeat (or rediscover) it.

Where do checklists fit in?

Security checklists cause both alarm (because they can replace thinking with conformance to irrelevant guidelines) and delight (because they serve as aids in checking that security considerations were properly taken into account).

The emphasis in higher education is on designing security in from the beginning rather than adding it afterward.

Checklists have their place in assurance. Consider the training of student pilots. They use checklists throughout the training process—everything from preflight (assurance that the aircraft is ready to take off) to in-flight maneuvers (assurance that all is well before beginning to bank or land). Professional pilots also use and respect checklists to ensure that neither over-familiarity nor hurry cause them to omit key safety elements. Checklists can be specific to individual crafts—the control panel on a Cessna 172 differs considerably from that on a Boeing 747—but a student pilot must be grounded in the principles used to develop these checklists in order to be able to generalize in emergencies. Students are required to study and understand the forces that affect the safety of aircraft in flight before they earn their pilots' licenses.

Taking the metaphor a step further, aircraft designers must have even deeper knowledge of issues involving aircraft safety, from structural considerations to the study of weather, and specialists who add to the body of knowledge surrounding flight must have still deeper knowledge.

This establishes a role for checklists with respect to secure programming. They can provide useful enumerations of specific actions required for safety and security. They should be derived from sound assurance principles, and then tailored to specific environments and protection needs. They aren't replacements for studying the principles driving their formation, nor should we consider them as substitutes for good judgment. Checklists can be used as examples of how we might implement principles, and they can supplement training by providing additional guidance for those who have been taught specific skills. However, they should not be used in isolation.

The ability to write secure code should be as fundamental to a university computer science undergraduate as basic literacy.

As the discipline of computer science matures, the ability to write secure code should be considered as fundamental to a university computer science undergraduate as basic literacy.

Teaching practice and principles can be intermingled in disciplines such as computer science, in which there is an inherent applied aspect. Prospective employers also have strong expectations that graduates will be proficient at both. Computer science undergraduates (and those who hire them) normally expect that students will graduate with the basic skills needed to obtain and hold down a job as well as an understanding of the concepts, principles, and methods of thinking that will let them remain proficient. Experience in security-enhanced programming is important to both of these goals, and certainly an asset in all areas of computer science.

There is growing concern in the academic community that the need for improved software security will cause pressure to teach skills focused on specific programming languages and operating systems at the expense of educating students in the important general assurance concepts. The function of academia is not to teach programming techniques, but to teach concepts, principles, and methods of thinking that students can apply to new situations. Further, it is incumbent on graduates and employers alike to support participation in ongoing professional training, so that employees can refine the specific skill sets needed to implement those concepts and principles within the context of their jobs. □

References

1. "Mars Climate Orbiter Mishap Investigation Board Phase I Report," NASA, Nov. 1999; see the press release at http://mars.jpl.nasa.gov/msp98/news/mco991110.html.
2. "Some Programs Seem to Stop Working after You Install Windows XP Service Pack 2," Article ID 842242, Microsoft, 8 July 2005; http://support.microsoft.com/kb/842242.
3. W. Ware, *Security Controls for Computer Systems: Report of Defense Science Board Task Force on Computer Security*, tech. report R609-1, RAND, Feb. 1970.
4. J. Anderson, *Computer Security Technology Planning Study*, tech. report ESD-TR-73-51, US Air Force, Electronic Systems Division, 1974.

Matt Bishop *is a professor of computer science at the University of California, Davis. His research interests include vulnerability analysis and denial-of-service problems, formal modeling (especially of access controls and the Take-Grant Protection Model), and intrusion detection and response. Bishop has a PhD in computer science from Purdue University. He is a charter member of the Colloquium for Information Systems Security Education (CISSE) and author of* Computer Security: Art and Science *(Addison-Wesley, 2002). Contact him at bishop@cs.ucdavis.edu.*

Deborah A. Frincke *is chief scientist of the CyberSecurity group at the Pacific Northwest National Laboratory. Her research interests include security of high-speed systems and system defense, especially intrusion detection. Frincke has a PhD in computer science from the University of California, Davis. She is currently on leave from the University of Idaho, where she is a professor and was director of the Center for Secure and Dependable Systems. She is a charter member of the Colloquium for Information Systems Security Education (CISSE). Contact her at deborah.frincke@pnl.gov.*