# Computer Security in Introductory Programming Classes

*Matt Bishop*
Department of Computer Science
University of California at Davis
Davis, CA 95616-8562

## 1. Introduction

The scope of computer security is as broad as the field of computer science; however, its most immediate impact to the average user is in the faulty, non-secure software, hardware, and systems that are deployed. The majority of these are designed, implemented and built, and fielded by people with much experience in their disciplines, but little in computer security. The basis of this position paper is the belief that to improve computer security education, we must increase the average computer programmer's understanding of the issues of computer security.

Like many facets of the field of computer science, computer security rests on a firm understanding of the basics of computer programming, such as design, implementation, testing, and deployment. In that sense, a student's first and second classes in programming are the beginnings of computer security education. This position paper argues that a major thrust of any initiative to improve the state of computer security education must begin with an analysis of teaching the basics of good design and programming.

For this paper, we do not consider what computer security classes per se should teach [1]. We instead focus on the role of general design and programming in developing secure systems.

## 2. Basic Programming Course Needs

In a basic (first or second) programming course, students focus on the art of programming as well as (or rather than) the mechanics: how to design a program, how to write robust, portable code, how to test the program, how to debug the program, how to analyze the code for perfor- mance problems, and how to write programs designed to interact with the particular system being used. Students also learn environmental issues such as careful software management (source code control systems) and software maintenance packages. Understanding the underlying principles behind these mechanics is central to understanding how to design and implement secure systems.

A good class will teach students not only to analyze these aspects of a program, but also to look beneath them at the assumptions of the environment in which the problem is to be solved. For example, if the problem is to write a trusted network server, the assumption in the requirements is that the operating system on the host is also trustworthy, for if not the system can lie to the server, which will make decisions according to bogus data. The programmer must either be prepared to handle the erroneous data in a safe manner, or else simply acknowledge that a compromise of the host will lead to a compromise of the server. The principles of robust programming as described below focus on this issue of trust, but it permeates every aspect of designing, implementing, and deploying secure systems.

### 2.1 Requirements, Specification, and Design

Deriving specifications from requirements and designing a program to meet those specifications is typically straightforward for basic programming classes, although it may require care. When security is a factor, the requirements involve security-related constraints, but the specification and design process is unchanged. Thus, undergraduate projects and exercises that emphasize the analysis of requirements (and derivation of specifications from them), and design (and design validation), help the student learn to design secure systems.

An important aspect of translating requirements to specifications is determining implied (as opposed to explicit) requirements. For example, if a program is to perform a function that changes between protection domains, there is an implicit requirement that the action be authorized in both domains, that the change of domains occurs in a secure manner (as defined by the site security policy) and that the change in domains is necessary. The programmer must translate these requirements (especially the second) into specifications.

Classes can teach any of several design techniques and skills. However, if the class teaches layered design approaches (top-down, bottom-up) the students should be able to validate each layer independently and then validate the interfaces; and if the class teaches the toolkit approach, the students must validate both the tools (for their particular purposes) and the connections among the tools.

### 2.2 Programming Skills

Learning to program is the most basic skill of a student of computer science and engineering. But like any other basic skill, it has many levels of complexity, and introductory computer science classes usually focus only on writing programs. Other levels may be mentioned but are rarely discussed in depth. They should be.

Specifically, security-related programming requires:

- modularity and information hiding; students should use modules to perform one set of related functions (in the style of reference monitors), and not allow other code access to the underlying data representations or the internals of the modules
- robust programming; students should use abstract representations of quantities manipulated by the modules, and should check for errors and handle them with care. The four principles of robust programming apply here:
    1. Be paranoid;
    2. Assume maximum stupidity;
    3. Don't hand out dangerous implements (hide internal data structures); and
    4. Worry about cases that "can't happen"
- portability; the student should know what parts of a program or system may depend on the particular compiler, host, or other environmental information, and should be able to write code that works correctly in differing environments.

Underlying these issues is the ability of the programmer to determine his or her assumptions. For example, in one version of a finger daemon, the programmers assumed that no user name would be over 512 characters long. By breaking these assumptions, attackers gain entry; indeed, the most effective attack technique is to read the manuals and try to violate any stated limits or error catching mechanisms. Programmers who know their assumptions can guard against the assumptions being invalidated, and the system being thereby exploited.

### 2.3 Testing and Debugging

Although testing and debugging usually occupy at least 50% of the development cycle, neither is taught well in the basic programming classes. Testing should demonstrate two aspects of the project: correctness (that the code meets the stated requirements and/or specifications) and robustness (that the code performs suitably given erroneous input or unexpected environments of execution). As part of the testing phase, programmers should check boundary conditions (especially for the off by one error), input validation, and any error conditions that the software or hardware tries to handle. In addition, any situation that can't happen should be simulated, to ensure that if the impossible does happen, the system responds acceptably. (Remember, programmers are invariably optimists.)

Recently, some raised the issue of formal (mathematical) derivation of programs. While a worthy goal, it seems to be to be far overrated, because the technology to formally verify programs is cumbersome at best, and most beginning programmers neither have the interest nor the mathematical depth to use verification tools effectively. This should certainly be taught elsewhere, but is inappropriate for a first or second course in programming.

## 3. Conclusion

Very few programmers learn the basics of good, solid programming. Most learn how to design programs to meet one or two requirements, and can write programs that work, but very few take the next step to develop robust code that is thoroughly tested and debugged, and that will work correctly in all relevant environments. This problem is endemic to our field, and needs to be addressed. Until it is, security problems will plague computer systems.

Finessing these problems will not work. One is tempted to say that programmers working on security-related programs could be licensed after they have taken special classes in this field, or have shown their competence in the above style of programming. Ignoring ethical and administrative issues, the problem is the definition of security-related. Any program is privileged when a privileged user uses it, and in the eyes of an attacker, a privileged user is any user whose privileges the attacker wished to acquire or exploit. In short, all programs are security-related under the right circumstances, so all programmers would need to be licensed and such a proposal is beyond the scope of this paper.

## 4. References

[1] M. Bishop, "Teaching Computer Security," *Proceedings of the Ninth IFIP International Symposium on Computer Security, IFIP/Sec'93* pp. 43-52 (May 1993).