Education
Editors: Matt Bishop, bishop@cs.ucdavis.edu
Cynthia Irvine, irvine@nps.edu

A Clinic for "Secure" Programming

Matt Bishop
University of California, Davis

Despite our reliance on software in everything from televisions and cars to medical equipment, it often doesn't work correctly. Everyone has had problems with software—text editors that freeze, causing us to lose work; answering machines that won't answer. These problems merely frustrate us. Others are far more serious, such as the program on a satellite that contains an error, causing the loss of expensive equipment, or Web servers that have bugs enabling an attacker to break in and steal personal data. In the case of medical equipment, poor software could cost lives.

So, how can we develop better software? One way is to make good programming as much a part of learning computer science as good writing is a part of studying English and law. So we developed a secure programming clinic to test this idea.

The Need for Robustness

Improving software quality requires many different approaches.[1] One is to give programmers the knowledge and skills to write programs that resist attacks and handle errors appropriately.[2] This style of programming is commonly called "secure programming," which implies the result is a program that can't be compromised or made to do untoward things. The definitions of "compromised" and "untoward" here both rely on knowing what security requirements the program is to enforce, which vary among installations. Underlying these definitions is the notion of robustness, which refers to properties common to all programs, such as not crashing and checking the length of inputs to prevent buffer overflows.

Principles of robust programming are widely taught in beginning and advanced programming courses. They're a large part of good programming style and require the realization that things can go wrong. A user might enter a string when the program expects an integer. A function that opens a file might fail. The program must check for these possibilities and handle them correctly. Grading in these programming courses takes such practices into account. But all too often in advanced computer science courses, students don't follow these practices.

This was true in our undergraduate OS course. In it, we require students to modify an OS kernel, such as Minix, to demonstrate their mastery of the covered concepts. The students work in teams and must demonstrate their modifications to the teaching assistant, who examines their code and tests it interactively.

One year, when grading the first assignment, the teaching assistant noticed that the programming was non-robust; many students didn't check inputs or assume that errors would occur, even though the modifications worked. So, he informed each team that he would deduct 20 percent of its points if the next assignment had code that was that poorly written. The students complained to the faculty member because "the code worked, and that's the issue!" The faculty member said he did disagree with the teaching assistant's assessment—the deduction should be 40 percent instead—but he'd leave it at 20 percent. All the code for the second assignment was very well written.

The students in this particular course all had taken introductory and advanced programming courses that emphasized good style—checking input values, handling function and system calls that return error codes, and so forth. In those courses, the students had to apply these stylistic concepts to get a good grade. But in most advanced computer science courses, grading is based on whether the student has correctly implemented the idea or methodologies being taught, not on the program's robustness. Naturally, students tend to ignore the latter. This leads them to believe that good programming practices are something special rather than a basic requirement of all programming.

A student drove this point home to me when I was teaching an introductory computer security course. The course's first two weeks review good programming practices by showing how to write programs and libraries that won't crash and will either meet their specifications or report errors. A later assignment is to write a program to

determine how a named user can access a file. The student received a very low grade because the submitted program did no error checking whatsoever; for example, if the file didn't exist, the program crashed. The student was outraged, pointing out that the assignment didn't specifically state that the program had to be able to handle missing files or other errors—"if you wanted us to do that, why didn't the assignment say so?"

The Problems with Grading Robustness

The obvious answer is to take robustness into account when grading the program. This introduces three problems.

First, as students advance in computer science, they must deal with more complex issues. For example, CS 1 courses don't discuss race conditions. Yet they're a serious problem in multiprogramming. So, the students' background knowledge (and, specifically, their progress in the program) affects their understanding of robustness, and they might need ancillary help.

The second problem is the focus of graders, teaching assistants, and faculty. Their main concern is that students learn the course material. So, they often grade using scripts, checking output against a reference output, and look only at the actual programming when the output is incorrect. Their focus on the main topics leads them to unintentionally de-emphasize those topics they expect the students to have already mastered—such as good programming style.

The third problem is the resources devoted to the course. Teachers use resources to help students learn the course material rather than help them practice what they're already supposed to have mastered and be using. This prioritization of resources is appropriate because a course's goal is to learn new material. But the students must practice good programming—otherwise, they fall out of the habit of using their robust-programming skills and, as I noted earlier, don't realize those skills' importance.

Improving Skills through Clinics

This suggests that continuous reinforcement of those programming skills would improve students' programming habits. Consider how law schools and many English departments handle the problem of students' writing. Writing well is a skill; continuous reinforcement improves students' writing abilities. But law courses and advanced literature courses focus on analysis and understanding. To help the students with and to reinforce good English and writing, they use writing clinics. These clinics aren't classrooms; they're places that review the student's writing and offer suggestions and feedback on grammar and exposition, without commenting on the content. The focus is on the means of expression, not on what's being said.

A similar approach for reinforcing good programming style overcomes the three problems I noted earlier. First, the clinician can take into account the student's background. If the student hasn't learned how to write robust programs, the clinician can suggest sources and help the student with the problems that the clinician uncovers. Because the clinic isn't tied to the material in a particular course, the clinicians need not worry about whether the program correctly implements B-trees; they can see whether the student handles null pointers properly. They can check that memory containing passwords and cryptographic keys is reinitialized when those passwords and keys are no longer needed. Finally, because the clinic is not a part of any course, it requires no course resources.

The Prototype Clinic

Using a small grant from the US Department of Defense, we implemented this idea in the computer security course.[3] The students received an assignment to write a program to check a number of attributes of a specific file. If the attributes matched certain specifications, the program would change the file permissions and the file's owner (many privileged programs do this when creating log or temporary files). The potential problem in this program is a race condition in which a second file with attributes that don't meet the specifications replaces the first after the check but before the change.

The assignment's rules required the students to submit the programs to the clinic at least a week before the assignment was due. The students then had to meet with the clinician that week to review the program. They could make changes before turning in the final version for grading.

The clinician examined 17 programs. The initial versions had many problems. Table 1 summarizes the seven

most common problems; the second column shows the percentage of the assignments that had them before the clinic.

Table 1. Common problems with students' programs, before and after the clinic.

Programming problem	Before clinic (%)	After clinic (%)
Race condition	100	12
Unsafe function call (strcpy, strcat, and so on)	53	35
Format string vulnerability	18	0
Unnecessary code	59	53
Failure to zero-out password	70	12
Failure to do sanity check on file modification time	82	35

In all cases, the clinician pointed out at most one instance of each type of problem in each program, and left it to the students to find others. (In some cases, he didn't point out all the types of problems because there were too many!)

After the students turned in the assignments, we reviewed them to see the changes students made. The third column of Table 2 lists the percentage of programs that still had at least one instance of the named problem. These results include problems introduced in the changes. For example, some students fixed unsafe function calls by replacing strcpy with strncpy but forgot to set the last byte to NUL. All students who initially forgot to zero-out the memory containing the password added code to do so—but some put the code at the program's end, just before it exited. All students with unnecessary code deleted some, but most didn't delete it all (they left in some unnecessary checking).

An interesting observation arose with the time check. Initially, 14 students didn't check the file modification time to ensure it met the specifications. The clinician pointed this out to four of the students. Of the other 10, four found it on their own after meeting with the clinician, and fixed it.

Many students modified their programs by simply commenting-out the old code and entering the changed code. So, we could see they applied the robust-programming principles and methods in places that the clinician didn't identify for them. This suggests that the clinic would be more effective if it continued over a longer time period.

We asked the students for an anonymous evaluation of the clinic. They felt the clinic reinforced robust-programming skills and were more confident that they could avoid the problems found in their programs. They also said that the interaction with the clinician (a graduate student who wasn't the course teaching assistant) made the problems clearer and easier to understand (and remedy) than if they had simply been given a set of rules to apply. All students thought the clinic was worthwhile.

Important Questions

These results suggest that exploring the use of a clinic would be worthwhile. Such an exploration would deal with several questions.

Should use of the clinic be mandatory? Ideally, students would use it when they felt they needed help. In our exercise, students had to use the clinic in order to have their program graded. A few didn't (because they didn't read the assignment or finish it in time). (Interestingly enough, these students still liked the idea of the clinic.) An alternate approach would be to make the clinic voluntary, which would make it resemble typical writing clinics. We required the clinic because experience suggests that, if it weren't required, many students wouldn't have used it. Because all the course assignments required robustness, we wanted students to do as well as possible in that area.

Should the clinic grade programs as well as help students? Certainly, the clinic shouldn't grade the programs when students bring them in for help. However, clinic personnel can relieve graders and teaching assistants of checking for lack of robustness by grading that aspect themselves. Furthermore, because of their experience, they can provide much more detailed feedback, freeing the course graders to focus on the aspects of the program relevant

to the material being covered. However, in our experiment, the clinician played no part in the grading.

Who should the clinicians be? We used a graduate student working in computer security who was familiar with robust-coding issues. An alternative would be practitioners who employ these skills in their work. This would emphasize to the students that these skills are important in nonacademic environments and that employers expect their employees to practice them. It also would provide an avenue for industry and government to help academia improve people's programming skills.

Programming clinics appear to have promise. Our first attempt was very successful. Such an approach will help emphasize to students that they must use good programming practices throughout their educational career, and indeed will habituate them to using it. This will solve one part of the problem of poor software.

References

1. B. Curtis, "Top Five Causes of Poor Software Quality," Datamation, 1 July 2009; http://itmanagement.earthweb.com/entdev/article.php/3827841/Top-Five-Causes-of-Poor-Software-Quality.htm.
2. *Computer Science Curriculum 2008: An Interim Revision of CS 2001*, ACM and IEEE CS, Dec. 2008; www.acm.org//education/curricula/ComputerScience2008.pdf.
3. M. Bishop and B.J. Orvis, "A Clinic to Teach Good Programming Practices," *Proc. 10th Colloquium Information Systems Security Education*, 2006, pp. 168–174.

Matt Bishop *is a professor in the Department of Computer Science at the University of California, Davis. Contact him at bishop@cs.ucdavis.edu.*

Abstract. Everyone has had problems with software. Some problems are particularly serious, such as the program on a satellite that contains an error, causing the loss of expensive equipment. So how can we develop better software? One way is to make good programming as much a part of learning computer science as good writing is a part of studying English and law. So we developed a secure programming clinic to test this idea.

programming, robust programming, security and privacy, writing clinics, software engineering, software engineering education

DOI goes here.