Proceedings of the 10th Colloquium for Information Systems Security Education
University of Maryland, University College
Adelphi, MD June 5-8, 2006

A Clinic to Teach Good Programming Practices

Matt Bishop and B. J. Orvis, *University of California at Davis*

Abstract – We present an approach to emphasizing good programming practices and style throughout a curriculum. This approach draws on a clinic model used by English programs to reinforce the practice of clear, effective writing, and law schools to teach students legal writing. We present our model for a good programming practices clinic, and discuss our experiences in using it.

Index terms – software assurance, secure programming, education

I. Introduction

The problem of non-robust and non-secure programming practices is pervasive. Academic institutions teach proper programming in introductory classes, but often by the time students enter advanced courses, the teachers have only enough resources to focus on the correctness of code. Ancillary properties, such as robustness and security, are overlooked by necessity.

The reasons are complex. One key reason is the amount of material in the computer science curriculum. A glance at the ACM Computing Curricula [1] shows how much material must be compressed into courses for computer science majors. The focus of courses, naturally enough, is on the material intrinsic to the course and not to ancillary issues. Students also reflect this belief. Most teachers who deduct points for non-robustness or poor commenting have heard the protest, "But it works!"

Writing programs that work is indeed the point. But what does "work" mean? The students who raise the objection noted above mean that it works for a particular set of inputs, in a particular (or set of particular) environments. Move the program from a home Linux system to one in a computer lab, and the program may fail. Type input that is too long, and it may fail. Modify the program slightly, and the new program will not work because the earlier version had a problem that was not apparent until the change. Whether a program "works" is relative. The goal of robust programming is to write programs that either work or fail gracefully.

The focus of this paper is: how can we improve the quality of programs that students write *throughout* their undergraduate and graduate work? If this is done, the good habits that the students learn will remain with them longer, and may help improve the state of software throughout the industry.

The next section discusses the notion of "secure programming" and an approach to incorporating it into classes that requires no additional class time. We then present a model of this approach, and discuss our experience in trying it. We emphasize that we are reporting *experience*, not a scientific evaluation of the method, because our data set is too small to extrapolate from. We close with some suggestions for others who try this method, and some further work and research that would either confirm or refute the effectiveness of our proposed method.

II. Background

We first present a model of "secure programming." Next, we discuss an approach to helping students improve writing that is used by many English and literature departments and law schools. This approach is the basis for our model.

A. Secure Programming

The term "secure programming" is bandied about freely, but it is misleading. A secure program is a program that satisfies a set of requirements that are labeled "security requirements." [2] These requirements may be detailed, laying out what the program is to do precisely. They may be vague, describing only a subset of actions, the rest being irrelevant to the security. For this reason, we avoid the term "secure programming."

For each class, and indeed for each assignment, the requirements given in the assignment state what the program is to do *in general*. But other requirements are implicit. For example, an assignment may not explicitly state that "if the input is invalid, print an error message describing the problem and terminate gracefully." But programs that crash on invalid input are badly programmed. We focus on this idea.

First, we define a set of characteristics of a bad program. These fall into two classes: robustness and security. We describe each separately.

Author's affiliation: Department of Computer Science, University of California at Davis, One Shields Ave., Davis, CA 95616-8562 USA

ISBN 1-933510-98-6/$15.00

Robust programming is "a style of programming that prevents abnormal termination or unexpected actions" [3]. This style has four central principles that distinguish it from more conventional programming:

1. *Paranoia*: the program or function should not trust anything it does not generate;
2. *Stupidity*: assume the user or invoker cannot read any manuals, and be prepared to handle incorrect, malformed, and invalid inputs;
3. *Dangerous implements*: keep internal data structures and functions hidden, so users and callers cannot alter or invoke them (accidentally or deliberately); and
4. *Can't happen*: handle cases you believe are impossible, because even if they are, someone who changes the program may make that case possible.

As an example of the application of these principles, consider a program invoking a system call. The program must check the return value of the system call, unless the failure of the system call is irrelevant to the correct functioning of the program. For example, if one closes a file before a UNIX program exits, the failure of the system call is irrelevant because the kernel will close the file on process exit. But if the program closes the file and then opens another one, the failure of the close system call means that one additional file descriptor is in use. That could result in the process not being able to open another file (if the file descriptor table were full and could not be expanded). In that case, the program should report the failure of the close system call.

These general principles support writing programs with the requirement that they act correctly when possible, and exit gracefully and with an appropriate error message when not. Beginning programming classes teach that programs should behave in that fashion, regardless of their intended function.

Secure programming, as indicated above, means different things in different situations. For the purpose of this paper, we will use the term "secure" to mean that the program will not add or delete privileges or information unless it is specifically required to do so. For example, a file deletion program may delete user-specified files and be considered secure; but if it deletes additional files, it is not. The characteristics of a secure program are that it handle overflow properly, not have race conditions, and so forth. We should note that the precise set of characteristics varies from expert to expert. We will discuss this further in a later section.

The reader will note that robustness and security, called in what follows "good programming style," are simply aspects of software assurance.

B. Writing Clinics

College students are expected to compose essays and write understandably. This is critical in majors requiring communication skills and literary analysis skills, but is also important in all majors. But institutions also recognize that many students do not have those skills when they enter. Even when they do, these skills must be refreshed continually, or they atrophy. Colleges provide the needed support in two ways.

The first is to measure competency upon (or before) entry. If the student's competence falls below a certain level, he or she must take a class in remedial English before taking the basic English classes required to graduate.

The second is to provide support for writing in classes. Many law schools, for example, offer "legal writing clinics" to help students improve their legal writing. Some English, rhetoric, and comparative literature programs have access to similar clinics to help students improve their writing. Universities sometimes make available similar clinics to students for whom English is a second language, regardless of their major. This method works by having students bring writings to the clinic. The clinic member reviews the writing for sound organization and structure, clarity, and (sometimes) grammar and spelling errors. The clinic member does *not* review the material for accuracy or completeness; the focus of the clinic is simply on the quality of the writing.

This second method does not add to the material in the curriculum. It is an adjunct technique for building on an existing foundation, and emphasizing its importance by aiding (and requiring) good writing throughout all classes that the student takes. The instructor(s) of the class(es) can focus on content rather than on English.

The parallel with computer science classes is straightforward. We want to support students who already *know* how to program, but who may not be adept at good programming style. We want the manner of support to be provided independent of the material discussed in class, so the support can be provided for any class in which programs are to be written. Finally, we want the instructor to focus on content and correctness of the program with respect to the particular requirements of the assignments, rather than on good programming style.

III. SECURE PROGRAMMING CLINIC

The genesis of the clinic occurred in 2002, when the author taught an operating systems class. One of his TAs was a graduate student in computer security, and he was assigned the task of grading several interactive labs that required the students to write programs and modify the

kernel for a small system, MINIX [4]. The TA graded content for the first lab, and then, with the students present, critiqued their programs for good programming style. He informed the students that unless they used good programming style, he would deduct 20% of their score next time. The result was that most students dramatically improved their programming style in later programming assignments. This demonstrated that students would respond to a requirement that they use good programming practices. Here, those factors influenced the grade directly.

We now change perspective slightly. We consider good programming practice to be foundational. It is something all students in computer science classes that involve programming are expected to know how to do and to apply in all programming assignments, regardless of the specific requirements of the class they are coding for. But institutions must recognize that many students do not acquire those skills in beginning programming classes, because they are learning the details of the programming language. Even when they do, these skills must be refreshed continually, or they atrophy. This suggests a programming clinic, analogous to the writing clinics discussed above.

Some assumptions about a student's background bear mentioning. In a writing clinic, the clinic members assume that the student has some familiarity with English (although the degree of fluency may vary wildly, especially if the clinic targets students for whom English is a second language). They will have writing skills commensurate with entering an English (or law) program in college. In the programming clinic, the clinic members can assume the students are familiar with some elements of good programming style, because certain basic elements are taught in all such classes. Students taking programming classes will have programming skills commensurate with a basic knowledge of programming. The parallel holds here.

In the programming clinic, the level of the students may constrain the security part of the clinic. A student in a software engineering course may not have taken an operating systems or programming languages course, and so may not be familiar with the concept of a race condition. This means that the people running the clinic must be sensitive to the background of the students they are helping, and may have to explain some problems (like race conditions) in more detail than others (such as buffer overflows). A similar problem arises in a writing clinic, because some students may know more about structuring and organizing an essay than others, so clinic members may have to explain some aspects and concepts of writing in more detail for some students.

A programming clinic may function in two different ways.

The first way is simply to assist students. The clinic members do not grade or evaluate programs. At student or faculty request, they examine a program that a student has written, and meet with the student to suggest improvements to make the program more robust or handle potential security problems (as described above). The student is then expected to take this information and make the improvements. Ideally, the student will learn from this interaction. Over a period of time, the student's style should improve.

The second way is for the clinic members to provide the instructors with a grade for the robustness and general security issues. This may involve no interaction with the students, for example the class graders checking that the program meets the specific requirements of the assignment, and the clinic members checking for non-robust and other types of problems. This may involve interaction along the lines of the first method, followed by grading as above.

Each method has advantages and disadvantages. The first method is strictly an adjunct to existing classes. The instructor may only change how assignments are handled, to provide students enough time to have the clinic check the assignments. If he or she prefers, the instructor can take advantage of the clinic by requiring students to have their programs checked. In this case, more time for the assignment may be necessary, especially if the clinic is small and the number of students is large. The disadvantage of this method lies in how the program is graded. If the instructor assumes the students will follow the clinic's recommendations, the students will have no external incentive to do so. But if the instructor wishes to reinforce what the clinic teaches, the instructor and graders must check for robustness and general security issues. They *should* check for this in any case, but in practice most graders will check simply that the program meets the specific requirements of the assignment.

The second method has the advantage of reinforcing what the clinic shows the student. It motivates the student to apply the lessons of the clinic upon pain of receiving a lower grade. The disadvantage is that the clinic becomes a part of the class, and the members of the clinic serve as graders for the class. Thus, they must take into account differences among instructors' grading schemes.

Whichever method is chosen, interaction with the students is critical. If the students simply submit programs and receive written comments, the benefits will be far less than if a clinic member can sit with a student and discuss problems interactively.

IV. An Experience

We tested this model in a class on computer security for undergraduates at UC Davis.

As part of that class, the instructor spends two days teaching the principles of robust programming, and applying them to a stack management library to show both poor programming style and good programming style. Another two days are spent on common security problems.

To ensure the students understood the material about common security programs, the second assignment asked them to analyze a small program for security problems and poor programming style, and the third problem had them apply Fortify's Source Code Analysis suite [5] to a larger program. In that exercise, they had to trace what an attacker would need to do to exploit some of the vulnerabilities that the SCA reported, and also to determine whether some of the reports were false positives.

The fourth assignment asked the students to write a program that checked a number of attributes of a particular file. If the attributes had the desired characteristics, the program changed ownership and permissions of the file. If not written properly, this program will contain a classic TOCTTOU file access race condition [6].

Before the program could be turned in, the students were required to submit it to the clinic. The graduate student manning the clinic would go through the program, and then meet with the undergraduate student and review the program with him or her for poor programming style. Once this review was completed, the student would modify the program as appropriate, and submit it to be graded.

Students were responsible for making the appointment with the clinic. The student needed to meet with the clinic members before the program was due. To do so, the student needed to complete a first version of the program that compiled and ran. The student sent the program to the clinic. The clinic member examined the program *before* meeting with the student. This enabled him to examine the program, and note potential problems that he could then discuss at length with the student.

Table 1 summarizes the problems with the programs submitted to the clinic. Seventeen out of 25 students participated in the clinic, and the number in the right column of the table indicates how many students had the programming problem in the left column.

programming problem	***number***
TOCTTOU race condition	100%
Unsafe function calls (*strcpy*, *strcat*, etc.)	53%
Format string vulnerability	18%
Unnecessary code	59%
Failure to zero out password	70%
Failure to do sanity check on file mod time	82%
Poor Style	41%

Table 1. Problems with programs mailed to the clinic.

The programs submitted for grading were substantially better than the ones submitted to the clinic. Most students attempted to fix the problems found by the clinic.

In the final program, 15 students (out of 17 who had the problem) fixed the race condition in the program they submitted.

Out of 10 students with the error initially, 8 fixed the use of unsafe function calls. Interestingly, 4 who replaced *strcpy* with *strncpy* did not set the last byte of the target to NUL. This is a problem only when overflow occurs, but it is a problem the students should have handled.

All 3 students with format string vulnerabilities fixed them. This was expected, as the fix for these is very easy.

All 10 students with unnecessary code deleted some, but 9 left some unnecessary code in the final program. Two of these involved unnecessary checking; the rest were either errors or calling an unnecessary system function.

All 12 students who did not zero out the stored password added code to do so. The purpose of this action is to prevent the disclosure of a password should the program crash and dump core. Solaris 2.6 had exactly this problem in its FTP server [7]. But 2 students put the code at the end of the program, rather than right after the password was validated. Thus, the location of the code defeated its purpose.

Fourteen students failed to add sanity checking on file creation/modification time. All 4 students to whom the problem was pointed out fixed it. Of the other 10 students, 4 found it after the clinic and fixed it. The specific sanity check required the student to check that the time recorded in the file's inode was not greater than the current time.

Many students commented out the original code when they modified the program as a result of the clinic's recommendations. The changes, including comments, showed the students seemed to absorb the lessons of programming style covered in the clinic. In this sense, the students' awareness of potential problems was heightened.

Students were asked to write a brief, anonymous evaluation of the clinic after they completed their final exam. All students viewed the clinic favorably. Those who participated felt that the clinic reinforced good programming practices, and many felt that the analysis done with the graduate student made the more abstract class presentation come alive. They felt more confident in their ability to write programs without race conditions. Those who did not participate wrote that they didn't check the assignment until too late or did not finish their program, but thought the clinic was a good idea. Amusingly enough, only 4 students admitted to not participating in the clinic; 3 who did not filled in the evaluations as though the had!

V. Recommendations and Future Direction

First, we present some observations. Then we discuss future directions.

A. Observations from the Experiment

What follows are generalizations from the experiences we had in running the clinic. We must emphasize these are not scientific observations, bolstered by formal metrics. These observations are based upon our experience, and comments from students in the class and the student manning the clinic.

The assignment in ECS 153 required that students submit programs to the clinic before grading. Failure to do so meant the program would not be graded. In retrospect, this was probably too harsh. Seven students did not do the program. In a post-course evaluation, they wrote that requiring the program be done a few days before the assignment was not feasible because of their schedules. One student suggested that the assignment should have two due dates. The first would be for an initial version of the program (much like a rough draft of an essay). The second version would be due sometime later, after the clinic members had reviewed it and the student made changes. How to grade an assignment done this way is an issue; perhaps the first program could be marked as running or not running, and the second graded on meeting the requirements of the assignment and general programming style.

A second issue is whether the programming practice help should come from someone associated with the class, like a TA, or from someone independent of the class. The benefit of the former is that the helper would know the assignment, and could point out places where the student did not meet the requirements of the assignment. This approach has the clinic member examining the content of the program. The benefit of the latter is that the helper focuses on programming practices common to all programs, not just those required by the class assignment. This approach has the clinic examining the structure and practices of the program. In our first use of the clinic, the graduate student was not associated with the class. He focused on the common programming practices, but in a few cases where he noticed students deviating from the requirements of the assignment, he pointed that out.

One issue that we encountered was student unfamiliarity with good programming practice in general. The subject matter of ECS 153 dealt with this problem because the first two weeks included lessons on both robust programming practices and common security problems. In other classes, this problem would be more severe. One approach to ameliorate this problem is to provide a handout describing the recommended practices. This handout should be succinct and tailored to the level of the students in the class. The clinic could reinforce the material in the handout in more depth, or the instructor could schedule a discussion section to do so.

B. Future Directions

In the short term, we have enough funding to run the clinic again in the Spring quarter, with one graduate student manning the clinic. The instructor for a freshman and sophomore level class, ECS 40 (called "Introduction to Software Development and Object-Oriented Programming") has agreed to have his students use the programming clinic for an assignment. The students will write the program in C, working in teams of one or two members. Students can met with the clinic any time before the program is due.

Because this class is the second programming class for computer science majors, the students will be solidifying the good programming practices they were taught in ECS 30, the first programming class. Further, they will not be familiar with race conditions and other potential security problems related to operating systems concepts. We will prepare a one or two page handout presenting the problems that the clinic will look for, and build a web page with additional information for the students to go to. The handout will provide high-level coverage, and have pointers to the appropriate web page or pages.

Beyond the Spring quarter, we want to develop and perform formal experiments to test the effectiveness of the clinic, and of the different ways the clinic can support instructors and students. As part of these experiments, we will need to increase the number of students manning the clinic, both to acquire more data and to provide better service. In particular, one person manning the clinic can only review so many programs. The size of the ECS 153 class was small enough to allow the clinic member to review all programs in a timely fashion. But ECS 40 will

have 80 students organized into between 40 and 50 groups. It is unclear how a class of that size will interact with the clinic member.

We believe the clinic has promise. It appeared to be a success with the students in ECS 153. Certainly the results of that experience warrant further testing. The advantages of the clinic, especially that it can work in parallel with classes to avoid the use of class time to teach robust and secure programming, may be enough to encourage its use to augment existing curricula.

Acknowledgement. The author gratefully acknowledges the support of grant H98230-05-1-0104 from the National Security Agency to the University of California, Davis. Thanks also to Fortify Software for making available a copy of its Source Code Analysis suite for the undergraduate class in computer security.

VI. References

[1] *ACM Computing Curricula 2001: Computer Science* (Dec. 15, 2001).

[2] M. Bishop, *Computer Security: Art and Science*, Addison Wesley Professional, Boston, MA (2003).

[3] M. Bishop, *Robust Programming*, handout for ECS 153, Computer Security (Jan. 2006); available at http://nob.cs.ucdavis.edu/classes/ecs153-2006-01/handouts/robust.pdf.

[4] A. Tanenbaum and A. Woodhull, *Operating Systems: Design and Implementation*, Second Edition, Prentice-Hall, Inc., Upper Saddle River, NJ 07458 (1997).

[5] *Source Code Analysis Suite*, Fortify Software, 2300 Geng Road, Palo Alto, CA; see http://www.fortifysoftware.com/products/sca.jsp.

[6] M. Bishop and M. Dilger, "Checking for Race Conditions in File Accesses," *Computing Systems* **9**(2) pp. 131-152 (Spring 1996).

[7] "Solaris FTP Core Dump Shadow Password Recovery Vulnerability," Bugtraq ID 2601 (CAN-2001-0421); available at http://www.securityfocus.com/bid/2601.

VII. Appendix: Clinic Guidelines

The following are a set of guidelines based on our experience with the programming clinic. We plan to use these for the Spring quarter. The term "clinicians" refers o the graduate students manning the clinic.

- Decide how much help the clinic will provide. Should clinicians verify that the assignment appears to be completed? Should they compile and run the program? Compiling the program ensures that the code is syntactically and grammatically valid.
- The clinic should set clear submission guidelines, such as requiring an archive or all the files needed to make the program run. This should include directions on building, installing (if necessary), and using the program.
- The clinicians should set a range of times to meet with students. Students should be required to meet with clinicians several days *before* the due date. Otherwise, they may want to meet a couple of hours before the assignment is due, possibly late at night.
- If part of a code is difficult for the clinician to understand, then it probably needs to be rewritten and properly commented. The clinician is a third party reviewer of the person's code, and one of the goals of robust and secure programming is to ensure that the code is safe and understandable. If obscurity in the code is necessary, the reason should be documented.
- When checking programs for a common assignment, the clinicians should keep records of problems they see (although not *whose code* they see it in). This will allow them to point out common gaps in students' backgrounds. This information can be given to the instructors of beginning programming classes (and other classes as appropriate) to strengthen the teaching. On assignments that have significant security concerns (such as the possibility of a TOCTTOU race condition), clinicians are likely to see the same problem recur in different students' submissions.
- The goal of the clinic is to improve general programming practices. Therefore, clinicians should recommend specific good programming practices such as using the safer string functions *strncpy*, *strncat*, *snprintf*, and others even if the student uses *strcpy*, *strcat*, and *sprintf* safely. This will ensure the student is aware of the potential problems should he or she use those functions in other ways. Similarly, they should point out standard library or system calls that simplify the program, and discuss any portions of the code that appear to be unnecessarily complicated. But they must also realize that some programming assignments *will* require students to do something in a complicated way.
- Clinicians will probably not catch all problems with student programs. But our goal is to promote good programming practices so that the

students will minimize security (and even non-security) problems by writing robust code.

- Ultimately, it is up to the student to take, or reject, the advice clinicians give.

ISBN 1-933510-98-6/$15.00 © 2006 CISSE