Some "Secure Programming" Exercises for an Introductory Programming Class

Matt Bishop

Some "Secure Programming" Exercises for an Introductory Programming Class

Matt Bishop[1]

[1] Dept. of Computer Science, University of California at Davis, Davis, CA 95616-8562, USA; *email* bishop@cs.ucdavis.edu

Abstract: Ideally, computer security should be an integral part of all programming courses. Beginning programming classes pose a particular challenge, because the students are learning basic concepts of programming. Thus, teaching them about buffer overflows as security problems, requiring an explanation of concepts such as "smashing the stack," will confuse students more than motivate them to check array bounds. Advanced concepts such as race conditions require more background than the students have, or will have, when taking introductory programming classes. An alternate approach is to teach the underlying concepts of robust programming; preventing crashes or errors is central to such a course. This paper presents some exercises that illustrate this approach, and some thoughts on what constitutes "secure programming".

Keywords: secure programming, robust programming, introduction to programming,

1. Introduction

Secure programming is a misnomer. A program may be secure under one set of conditions, and yet be woefully non-secure under a different set of conditions. As an example, a program that limits access to a resource to a few specified, authenticated users is secure when the goal of the system is confidentiality, but not when the goal is accessibility and the resource intended to be publicly available. For this reason, introductory programming classes should focus on much more concrete properties of programs: *robustness* and *correctness*. One can then introduce "security" in a later class as a collection of necessary (or desirable) properties for correctness, and the students will have the background to be able to focus on the *security* issues, knowing the robustness and correctness issues.

A second aspect of secure programming lies in design. When presented with a problem, students often try to program the most direct solution. Sometimes, a few minutes' thought will lead the student to a much simpler, easier program. The

trick is learning how to look beyond the statement of the problem to what the problem is trying to solve. Like understanding unstated meanings in talking with other humans, being able to see the essence of a problem may make the problem much easier—or point to a deeper problem that must be solved.

Section 2 discusses several programming exercises for introductory classes. These exercises illustrate various aspects of robust programming. Section 3 presents a problem that appears straightforward, but has an interesting subtlety of design. Section 4 discusses some aspects of a programming language suitable for introductory programming classes. Section 5 presents some concluding thoughts.

2. Robustness

Robust, or *bomb-proof*, programming is a style that prevents programs from acting unexpectedly, for example terminating abnormally. The basic principles of robust programming are:

- *Paranoia*. If your program or library doesn't generate it, don't trust it.
- *Stupidity*. Assume the user or caller won't understand your interface, and will send anything through it.
- *Dangerous implements*. If any data structure is visible to the user or caller, assume it will change between references.
- *Can't happen*. If you are sure it can't happen, check for it and return or print an error.

These basic principles underlie a myriad of security problems. As an example, the CWE/SANS Top 25 Most Dangerous Programming Errors [1] cites 9 weaknesses that arise from insecure interaction between components—in other words, by sending incorrect data through the interface, violating the principle of stupidity. The OWASP Top Ten project [2], which identifies the most serious web application vulnerabilities, cites cross-site scripting as the top problem and injection flaws (such as SQL injection) as the second most common problem. Both of these involve ignoring the principle of paranoia because it is trusting data that the program did not generate or check. Students who learn these principles and practice them are much less likely to create software with these problems.

The problems below were given to several classes during a first course in C programming. They can easily be adapted to work with other languages.

2.1. Demonstration of the Problem

One of the ways to impress upon students how important these principles are, and how often they are violated, is to give them an exercise that demonstrates problems with standard functions and libraries.

Problem. Please write three programs that use functions from the standard I/O library. You are to call the functions in such a way that they cause the program to crash, or generate unpredictable results. To demonstrate crashing, use output from *gdb*(1) to show that the crash occurred within the standard I/O library. To demonstrate unpredictable results, run your program without changes on at least two different types of computers in the student laboratories. Note that you ***must*** supply the correct type of argument for the function. You may not, for example, pass a character pointer when a file pointer is expected.

This problem is typically given near the end of the first course in programming, when the students have a basic knowledge of debugging, have used *gdb* to find bugs in programs, and have worked with the standard I/O library.

The (admittedly anecdotal) responses to this question are interesting. At first, the students are nervous because they don't believe they will find anything. Then one or two students will find something, and suddenly many of the students will become excited, and tackle the problem. It generates quite a bit of discussion, especially about the assumptions that the authors of the library made, and the environment for which the library was designed.

About 10 years ago, this problem was surprisingly easy; calls with **NULL** file pointers, or negative numbers, worked like a charm. Recently, though, the robustness of many versions of the standard I/O library has improved, increasing the difficulty of this problem. Thus, now this problem would probably be more suited for a second course in programming. For the introductory course, other libraries provide the (lack of) robustness required for this exercise.

The complement of finding errors is preventing them. This is the topic of the next exercise.

2.2. Handling Procedure and Function Errors

This problem puts the students in the position of the programmer, and has them program defensively. It deals with converting a string to an integer, a topic that causes problems because the students must deal with many possible errors.

Problem. The function *atol*(3) takes a pointer to a character string as an argument and returns the integer value corresponding to that string. Unfortunately, it has several problems:

- Overflow is not detected;

- If the string is not a valid integer, it converts as much of the string as it can and then stops; and
- Most implementations do not handle a leading "+" sign.

Implement a new function called *natol* (for *n*ew atol) that handles all these problems. Your function must have the interface:

```
long natol(char *numstr, int *errcode)
```

where *numstr* is a pointer to the string whose integer value is to be returned, and *errcode* is a pointer to an integer variable which, when *natol* exits, has one of the following values:

0. no error has occurred; or
1. *numstr* points to something that is not a valid decimal integer (this includes *numstr* being **NULL**); or
2. overflow occurred, and the number being read was positive. The result of the function is undefined (that is, you can make it return anything you like, but it must return *something*); or
3. overflow occurred, and the number being read was negative. The result of the function is undefined.

This problem teaches students how to handle errors within library functions. The functions should communicate the error back to the caller, so the caller can handle the error appropriately. Having the library write an error message to the standard output or error (or some other I/O stream) without documenting that side effect can cause serious problems should the caller be producing output in a particular format.

This problem also has a design aspect to it. Checking for overflow in a language-independent way is not at all obvious, especially to beginning students. The obvious approach, checking succeeding values until the absolute value of one is smaller than the absolute value of its predecessor, doesn't always work. The correct technique uses division. As this routine iterates over the characters in *numstr*, it appends one digit per iteration; thus, the check need only confirm that the digit being appended does not cause overflow. The key idea is not to append the digit; rather, it is to determine the maximum digit that *could* be appended without causing overflow. The students have to be careful programming this to avoid causing overflow when checking for overflow.

2.3. Handling Input Errors

A third problem extends the idea of checking for errors to user input by building on a common exhortation in C programming: avoid the use of *gets*(3), an input function known vulnerable to buffer overflow. Students are taught to use the function *fgets*(3), which requires a parameter indicating the size of the array in which the input line is to be stored. The subtlety of *fgets* is that if the line is too

long, only that part of the line that fits into the buffer is read. The next invocation of *fgets* begins reading where the previous invocation left off. This exercise provides a more intuitive interface. The function either provides the entire line, regardless of length, or (for backwards compatibility) can act like *fgets*.

Problem. Write a C function called *dyngets* that reads an input line of arbitrary length from a given file descriptor. The interface to *dyngets* is to be:

```
char *dyngets(char *buf, int n, FILE *fp)
```

On entry, `if buf` is not **NULL**, then this function acts exactly like *fgets*(3).

If `buf` is **NULL**, then it and the second parameter, `n`, are ignored. On exit, *dyngets* returns a pointer to an internal buffer containing the input line. This internal buffer is allocated using *malloc*(3) or *realloc*(3). If the line is too long to fit in the currently allocated internal buffer, the buffer is grown to be long enough to hold the full line. The return value is **NULL** on end of file or error.

You are to allocate the internal buffer for *dyngets*, and you must reuse the internal buffer whenever *dyngets* is called. This buffer should be allocated using *malloc* on the first call to *dyngets*, and as you read longer and longer lines, use *malloc* to allocate a new buffer and *free*(3) to free the old one, or *realloc* (with appropriate error checking) to change the length of the buffer.

This exercise requires students to manage memory in a way that is invisible to the caller. Common mistakes are to make the internal buffer visible externally, violating the principle of dangerous implements, or not properly handling the internal buffer by either deallocating it at the beginning of each invocation, or failing to check the return value of *realloc*.

A second subtlety arises from the emulation of *fgets*. As the second argument is an integer, students must check that it is a non-negative integer. Although *fgets* should check this, some versions do not, and this causes unpredictable behavior.

2.4. Assuming the Obvious: Does 1 == 1 Always?

Students who first encounter floating point numbers do not realize that they represent a subset of the set of real numbers, and that the differences can adversely affect calculations. Some real numbers can be expressed exactly (such as 1/2), but others have no such floating point representation (such as 1/7). This exercise poses the question of "what does a floating point 1 represent on a computer".

Problem. Every computer is limited in the amount of precision it can represent for floating-point numbers. At some point, where `epsilon` is very small, the following expression will be true:

```
1.0 == 1.0 + epsilon
```

Write a program to find the largest value of `epsilon` on your computer for which the above is true. Note that the value of `epsilon` may be different for floats and doubles. Find both values (and the value for long doubles if your compiler supports them).

This exercise has two effects. The first is to show students why one needs to question assumptions. Unless students understand how floating point numbers are represented (a topic often omitted in introductory courses), the discovery that *epsilon* can be non-zero helps them understand the need to follow the principle of "can't happen". Something that appears to be wrong is, in fact, correct!

The second is an interesting design question. Some students will treat the expression as an *algebraic* expression, and instead attempt to find the smallest value for *epsilon* that is equal to 0.0. That approach fails because very small floating point numbers can be distinguished from 0, but when added to 1, the precision of the floating point number is reduced considerably. That is, most computers can represent the real number 2^{-63} exactly, but cannot represent the real number $1+2^{-63}$ exactly, due to limits on the size of the mantissa. So the algorithm the students design must take this into account.

2.5. Summary

This section presented four problems that require students to take care to avoid non-robust behavior: incorrect results or program crashes. The first demonstrates that system libraries, on which programs rely, may be non-robust. The second and third problems encourage the students to apply the principles of robust programming to make their library routines "solid". The fourth challenges a simple yet common idea among students taking an introductory programming class: that computers are precise and exact. In fact, they are not, particularly when dealing with floating point numbers.

Two of these problems had design components. We now focus on that aspect exclusively.

3. Design

Robust programming favors simplicity and elegance of algorithm. It is much easier to avoid unnecessary interfaces and poor coding when the program is simple and straightforward. Often, complex or long problems have simple solutions, and when they do, spending time to find that solution is well worthwhile.

The Monty Hall problem is a wonderful problem for teaching principles of robust design. It is based on the old TV game show *Let's Make a Deal*. In that show, the moderator, Monty Hall, would select a member of the audience, and offer them a valuable prize. The prize was behind one of three doors. Behind the other two were joke prizes, like a goat and a can of paint. The member of the audience would select one of the doors (say, door number 2). Monty would then

say, “Before I show you what’s behind door number 2, let me show you what’s behind door number 1.” Door number 1 would open, to show the can of paint. Then Monty would ask if the player wanted to change to a different door. The question is, should the player do so?

The intuitive answer is that it doesn’t matter. As the door Monty opens always has a joke prize, one of the two remaining doors has a joke prize, and the other has the valuable prize. Hence the valuable prize is equally likely to lie behind either door.[1]

Problem. Write a program to simulate 100,000 iterations of the Monty Hall problem. Use the simulation to demonstrate whether it is to the player’s advantage to change doors. Please explain your results.

When given in an introductory programming class at UC Davis, student assignments used essentially the following algorithm (iterated the requisite number of times):

```
choose a random door for the valuable prize
player chooses a random door
Monty shows player a door without the valuable prize
generate a random number between 1 and 2 inclusive
if that number is 1, player changes door
if player’s door is same as door with valuable prize, increment counter
```

At the end, the program divides the counter by the number of iterations for the probability that the player will get the valuable prize should she change doors.

A much simpler solution arises when one notices that the number of the door Monty shows is irrelevant. It will *never* be the one with the valuable prize. So, all that matters is whether the player’s initial choice is the door with the valuable prize. If it is not, the switch will give the player the valuable prize. If it is, the switch will give the player a joke prize. Thus, the following algorithm also solves the exercise:

```
choose a random door for the valuable prize
player chooses a random door
if player did not choose door with valuable prize, increment counter
```

The probability is computed as before.

The reason this problem is so useful is because the correct answer is counterintuitive, unless you look at the problem in the right way; and the simulation appears to give the wrong answer. Thus, this problem forces the students to analyze their design in detail.

---

[1] The correct answer requires the student to remember there are *three* doors, not two. The probability that the initial selection is the door with the valuable prize is 1/3. The probability that the valuable prize is behind one of the other two doors is 2/3. When Monty opens a door, that door cannot contain the valuable prize. Thus, the probability is 2/3 that the unselected, unopened door is the door with the valuable prize. So it is to the player’s advantage to change her selection.

While this program is not, strictly speaking, security-related, it teaches the students skills they need for secure programming. They learn how to ask what the goal of the problem is and focus on meeting that goal, rather than simply choosing the obvious approach. They also see what happens when the mechanism does not work as expected; they learn to question the expected result and analyze it, to see if it is in fact correct. This skepticism is critical to being able to determine the assumptions that a program relies upon—a key aspect of secure programming.

5. Conclusion

The rudiments of secure programming lie in the first programming classes all students take. By emphasizing careful design, robustness, and correctness, those courses can lay a foundation upon which advanced classes can teach programming that deals with specific security problems. Without such a foundation, all the instruction into how to code securely will do little to improve the state of software security.

Acknowledgements. This paper is an expanded version of a working paper given at the Secure Coding Faculty Workshop sponsored by the National Science Foundation and SANS held in April 2008.

References

1. Christey, S.: CWE/SANS Top 25 Most Dangerous Programming Errors. http://cwe.mitre.org/top25 (Mar. 10, 2009).
2. Williams, J. and Wichers, D.: Top 10 2007. http://www.owasp.org/index.php/Top_10_2007 (2007).