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

Teaching Security Stealthily

Matt Bishop
University of California, Davis

Many people and institutions are considering how to make students who aren't taking computer security classes aware of the issues and make them think about how to improve the state of software and systems. Elementary programming classes teach problem-solving skills as well as programming-language syntax and semantics. Database classes focus on the theory and practice of storing, viewing, and extracting data. Algorithm classes emphasize algorithm design and analysis. Their teaching objectives might not even mention security. A class's topic constrains its content, and any information or practice of security is strictly incidental to the students' understanding and internalization of the material relevant to the topic.

However, information assurance and computer security are cross-disciplinary topics. They arise in all computer science disciplines. Thus, their material can and should be integrated into existing classes—but doing so presents a conundrum. Already, computer science classes' curriculum covers more material than can be comfortably taught in each class. So, rather than remove material about the class's primary topic, classes can introduce security-related concepts in homework exercises, thereby emphasizing those concepts in the context of that topic.

Here, I present four exercises: one from a class introducing nonmajors to computers and programming, and three from a beginning programming class for majors. We focus on introductory classes because they form the basis for students' computing experiences. Introducing concepts of information assurance and computer security early makes the students more aware of these issues.

Introduction to Computers

This class covers a computer's basic workings and introduces programming concepts. Students learn to write a simple Python program and, in doing so, learn about variables and assignments, conditional statements, loops, and input and output statements. They also learn about programming errors such as nonterminating loops. We added exercises to teach students how to handle problems arising from interactions with the environment, such as reading a string when an integer is expected.

In one sense, checking input is simply good programming practice. But it also might be a security issue, as noted in extant lists of security vulnerabilities (see http://cwe.mitre.org and www.us-cert.gov/cas/techalerts) and programming problems (see http://cwe.mitre.org/top25). In any case, it's clearly an issue of information assurance; if the information is bad, the computation's results will be bad too. So, teaching students how to check input is important both as a programming practice and for avoiding security problems. The specific concept is, *validate input before use*.

One assignment for this class is

> *A grocery store has hired you to write a program to help its checkout clerks make change. The teller will enter the amount of change to give the customer. Your program is to print the number of $100, $50, $20, $10, $5, and $1 bills and the number of quarters, dimes, nickels, and pennies that the clerk is to give the customer.*

During testing, the graders enter two deliberately erroneous values. The first is an amount like 35.672 (note the extra 2); the second is the string "one hundred dollars." The program is expected to give error messages in both cases: "too many decimal places" for the former; "amounts must be numbers" for the latter.

The first error is simple to handle in any programming language, provided the students use integer computations. The second requires some finesse in many programming languages. In Python, however, the easiest way to handle this case is to simply read the input and convert it to a floating-point number. If the number isn't valid, the conversion raises an exception that the programmer catches with the `except` statement. (Incidentally, such easy exception handling is one of the benefits of teaching Python.)

This exercise teaches the students about bad input and exception handling. Because students with little to no programming experience take this class, we give this assignment in two parts. The first, given immediately before we discuss exception handling, lets the students assume all inputs are valid. The second part, given right after, has the students modify their programs to handle invalid input. This lets them focus on exception handling. From then on, we require them to handle bad inputs in all programs, whether or not the assignment explicitly directs them to do so.

One question is whether to teach the students to use the exception rather than have them check that the input string is a number. Using the exception has two advantages: it's easy, and it generalizes to many types of problems. So, it covers multiple topics at once. However, students who later learn languages such as C or C++ will be at a disadvantage because those languages don't use exception handling in this way. Because this course is for students who aren't going to take programming or any other computer science class, the advantages outweigh the disadvantage.

When we give this exercise, we don't tell the students that it's related to security. Instead, we ask how many of them have ever mistyped something when entering it into a computer. Because most nonmajors are familiar with computers and programs that handle input mistakes poorly, using this justification places the problem (and the need for a solution) in a familiar context. We could do the same with problems more obviously related to security, but those typically require more complex programs than we ask the students to undertake in this class.

Introduction to Programming

This class delves more deeply into programming than the nonmajor class, so we can use more sophisticated assignments to emphasize security concepts. Here we examine questioning (usually implicit) assumptions, and some forms of integer overflow.

Questioning Assumptions

Arithmetic with floating-point numbers offers fertile ground for erroneous assumptions. Perhaps the most common is performing checks for equality—for example,

```
if (x == 1.0)
  … do something
else
  … do something else
```

Unless the value 1.0 is a sentinel, this conditional will usually result in "do something else" executing, because of the problems of round-off and numerical error. The following problem forces the students to think about errors:

> *Find the largest number ε such that 1.0 == 1.0 + ε.*

We give this exercise before we discuss floating-point representation. The results show the students that something they initially think can't be true (after all, algebraically, ε must be 0) is in fact true. It helps them break the connection between the abstract mathematical construct of equality and the applied programming construct, which is constrained by the computer's limits.

We discovered an additional, unexpected benefit. Some students treat the problem as an algebraic equation. However, instead of simply asserting that ε must be 0, they try to find the largest value for *epsilon* that satisfies the comparison

ε == 0.0.

This approach's problem is that the computer can distinguish very small floating-point numbers from 0, but when you add such a small number to 1, the floating-point number's precision decreases considerably. For example, a computer might be able to represent 2^{-63} exactly, but it can't represent $1 + 2^{-63}$ exactly, owing to the mantissa's representation. This is an example of you might call Needham's Law of Optimization: "replacing something that is expensive but works with something that is cheap and 'sort of' works."[1]

Compared to other approaches, this one is easier and, on most processor architectures, cheaper, but for this particular problem, it's also wrong. So, students learn to think through a transformation's implications and to not assume that two mathematically identical problems are computationally identical.

Ultimately, this problem challenges two assumptions of most beginning programmers: that mathematical equality means computational equality and that mathematically valid transformations are also computationally valid. By showing these assumptions' failure, we provide a basis for students to question other assumptions—and much of security is simply ensuring that assumptions are correct.

Checking for Integer Overflow

The presentation of integer arithmetic often fails to note that overflow can occur, resulting in unexpected or corrupt results. The following exercise combines an overflow with an obvious, but incorrect, solution:

> *Determine whether $a * b$ will cause (integer) overflow.*

When students first confront this problem, they usually compute the product. If the absolute value of the result's representation is less than the absolute value of the multiplier or the multiplicand, they conclude that overflow has occurred. For example, on a 32-bit system, $2^{30} * 2^{25} = 2^{55}$, which becomes 0. Because 0 is less than either operand, overflow has occurred.

However, the product's representation can be less than either operand when overflow occurs. For example, $(2^{16} + 2^{15}) * (2^{16} + 2^{15}) = 2^{33} + 2^{30}$, which overflows in a 32-bit system and produces a product of 2^{30}. And $2^{30} > (2^{16} + 2^{15})$.

Most students opt to compute the product probably because the results are true for simple cases. But with more complex cases, this test's incompleteness becomes apparent. Because the question is whether the multiplication operation works properly, the test shouldn't use multiplication. In what follows, we assume that *a* and *b* are positive integers.

We want to determine whether the product $a * b$ is greater than the maximum integer that the system can represent. Call this integer *MAXINT*. If $\lceil MAXINT/a \rceil < b$, the product will be greater than *MAXINT*. Otherwise, it won't. Furthermore, because these operations involve quantities no greater than *MAXINT*, overflow won't occur.

This exercise teaches the importance of thinking carefully when creating checks for error conditions. It also teaches that the obvious approach is often wrong and that when developing those tests, you need to consider boundary cases as well as the majority of cases. Both these concepts are critical to understanding how to write good programs and (applied more broadly) how to develop and test security mechanisms in general.

Rollover

Another common problem occurs when a counter overflows, which typically causes the counter's value to become 0. A good example of this is the Year 2000 (Y2K) problem, in which the counter representing the year overflowed its allocation of two decimal digits. A problem to demonstrate this rollover applies the Y2K problem to Linux- or Unix-based systems:

> *Determine the time (to the second) when the FreeBSD clock overflows. What are the latest date and time before the overflow, and the earliest date and time after the overflow?*

This problem surprises many students because they have never thought of how time is represented on a system—and those who have often believe the problem has been solved for all systems. In fact, many Linux- and Unix-based systems store time simply as a 32-bit signed integer. The students must experiment a little to discover this, but once they have discovered it, the solution is straightforward.

This exercise combines the questioning of assumptions with overflow. It also shows the students the consequences of ignoring possible overflow. In addition, you can make a direct connection between the time that systems enter into the audit log and the ability to understand when events happened—a key issue in security. Finally, it amuses most students to learn that, whereas the Y2K problem has been solved, the Y2K + 38 problem is still around.

Making security a part of exercises has several benefits. First, it minimizes the introduction of new material in the classroom. Second, students learn by doing, a far more effective method of acquiring knowledge than simply reading or listening. Third, students enjoy challenges, and security issues create challenging problems because the students must think of unusual cases in which the obvious solutions might be incorrect. This also teaches them to question assumptions (which many students eagerly embrace when framed as a challenge to authority), a key security concept. Finally, students learn to handle unusual cases or situations, something that serves them well not just in understanding security but also in designing and implementing systems.

In an earlier article in this department, Prof. Kara Nance presented projects in introductory classes that asked students to deal with security problems such as file recovery and printer forensics.[2] These require integrating security material into the class curriculum. The approach suggested here is less direct but perhaps easier, because the exercises involved are small problems. Which method works better, or whether a combination of methods would best suit a particular class, is for the instructor to decide.

References

1. R.M. Needham, "Denial of Service: An Example," *Comm. ACM*, vol. 37, no. 11, 1994, pp. 42–46; http://cs.gmu.edu/cne/modules/acmpkp/security/texts/DENIAL.PDF.
2. K. Nance, "Teach Them When They Aren't Looking: Introducing Security in CS1," *IEEE Security and Privacy*, vol. 7, no. 5, 2009, pp. 53–55.

Matt Bishop *is a professor in the Computer Science Department at the University of California, Davis and a co-director of its Computer Security Laboratory. Contact him at bishop@cs.ucdavis.edu.*