EMBEDDING CYBERSECURITY IN THE SECOND PROGRAMMING COURSE (CS2)*

Michael Verdicchio, Deepti Joshi, Shankar M. Banik
Department of Mathematics and Computer Science
The Citadel
Charleston, SC 29409
843 953-5039
mv@citadel.edu, djoshi@citadel.edu, baniks1@citadel.edu

ABSTRACT

Cybersecurity is currently a key thrust area for public, private, and governmental sectors. In order to protect our society from cyber-terrorism, cyber-espionage and cyber-warfare, we need to change the mindsets of our current and future workforces. The key to doing so is to embed cybersecurity education throughout the computer science curriculum. Rather than relegating the principles and practices of cybersecurity to only the few students who are able to take high level technical electives in those topics, this approach brings cybersecurity education to all of the students who may take one or more computer science classes. In our previous paper we developed our methodology of embedding cybersecurity concepts in the first computer programming course (CS1). In this paper, we continue our work on embedding cybersecurity concepts across the computer science curriculum, and focus on the second computer programming course (CS2).

INTRODUCTION

Cybersecurity focuses on protecting computers, networks, programs, and data from unintended or unauthorized access, change or destruction. The impact of cybersecurity affects all aspects of computing starting from programming, to data structures, networking, databases, architecture, and software engineering. In response to the increasing demand for a more cybersecurity-aware workforce, recent models for cybersecurity education design cybersecurity courses for junior/senior level students and

* Copyright © 2016 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.
offer them through either a minor program or a concentration. This implies that cybersecurity is not a required component of the computer science curriculum and has the drawback that students are graduating with computer science degrees but without having knowledge in cybersecurity. As a result, they are more likely to produce software replete with security vulnerabilities due to poor programming practices. It is our claim that cybersecurity simply cannot be treated as an added component to the computer science curriculum. In order to change the thought process of computer science students, these critical topics should be embedded throughout all computer science courses. For example, when students learn how to code in the introduction to programming class, they should also be taught to think about the security implication of each program they write.

Background

In 2010, President Obama established the National Initiative for Cybersecurity Education (NICE). NICE is a nationally coordinated effort focused on cybersecurity awareness, education, training, and professional development. It seeks to encourage and help build cybersecurity awareness and competence across the nation to build an agile, highly skilled federal workforce capable of responding to a dynamic and rapidly developing array of threats.

Under the NICE framework and other CNAP (Cybersecurity National Action Plan) educational initiatives, it is clear that the strongest action of all is the right education for the users and developers of the technology that can potentially come under attack. It is important to change the mindset of future generations with respect to cybersecurity, and make it an integral part of their computer science education. However, under the CS2013 curriculum guidelines proposed by the Joint Task Force from ACM and IEEE [4], the guiding principles do not include secure programming. Although the new content area on Information Assurance and Security (IAS) has been added, it is not clear on how to embed it within the CS curriculum [2, 3].

While there has been commendable progress in creating instructional modules to inject cybersecurity concepts throughout CS courses [5], further innovation is required to hasten adoption by a diverse range of instructors and faculty, many of whom lack cybersecurity experience or training. In an effort to make the incorporation of these concepts as easy as possible, we avoid the approach of stand-alone modules or lessons that must take the place of what is currently being taught in CS courses. Instead, by way of code examples, we provide a mapping of cybersecurity concepts to topics already being taught in our courses. In this way, instructors can adapt what they are already teaching to include essential cybersecurity concepts currently lacking exposure across the curriculum.

Project Goals

In our previous work [1], we contributed a mapping of cybersecurity concepts to topics traditionally taught in the first programming course (CS1). The units/chapters mapped include – 1) System Development Life Cycle, and Introduction via "Hello
In this paper, we build upon this work and contribute mappings of cybersecurity concepts to topics traditionally taught in the second programming course (CS2). Rather than providing modules or lessons that must be substituted into an already planned and busy course calendar, these examples are designed to be incorporated into topics already being taught to bring cybersecurity awareness to the forefront across the CS curriculum whether or not the instructor has cybersecurity experience.

**METHODOLOGY**

In this section we provide five code examples mapping to topics typically taught in the second programming course (CS2) for institutions offering a Computer Science degree or certificate. The examples provided here are in the Java programming language, but can be easily implemented using C++, Python, or other languages.

**Chapter 1: Methods**

While introduced in CS1 courses, the topic of object-oriented class design is typically reinforced and mastered in CS2 courses. Students typically model real world entities, such as employees, students, or bank accounts, as classes from which objects can be instantiated, and actions, such as withdrawing and depositing money to bank accounts as methods. When the emphasis is solely on object-oriented design, however, good security practices, such as the validation of method parameters, can be overlooked. When a method is implemented, it is important to validate the parameters of the method before they are used in expressions inside the method. In Figure 1, a deposit method for a typical BankAccount class is provided. The method adds the parameter amount to the balance of the account. The method does not check whether the input parameter is a negative value, which equates to a withdraw action instead of deposit.

*Security implication:* Attackers try execute methods by passing all kinds of input parameters in order to make the method behave abnormally. If a program is written where methods do not validate input, the program is vulnerable to an exploit.

**Chapter 2: Inheritance**

Any non-final class can be extended, and this point is sometimes not emphasized while reinforcing the concept of inheritance in CS2 classes. Even if instance variables remain private, the behavior of any public method can be overridden in the child class and produce misleading or unintended functionality. Through polymorphism, child class instances can be referred to by any parent type reference. Let us consider the rest of the BankAccount class from Example 1 in Figure 2. The toString method references a public getter method, getBalance. Since this class does not prevent inheritance, this
method could end up invoking an overridden version of `getBalance` and display inaccurate results.

**Security implication:** Unintended behavior might be implemented by a child class. An example of such method overriding from a malicious child class (called `SneakyAccount`) is shown in Figure 3, where the `getBalance` method now returns zero in all cases. The `toString` method inherited from `BankAccount` will invoke this version for all child class types. The other method allows us to see what `getBalance` should have returned had it not been overridden.

In figure 4 we show how a malicious child type can misrepresent object states if inheritance is not prevented. Suppose a vulnerability exists in `BankAccount` which allows withdrawals of negative numbers, creating deposits. Wishing to avoid attention, the child type `SneakyAccount` is created and overrides the `getBalance` method to always report $0. The exploit is performed on a normal account and for the malicious type, and when the account reporting procedure is run, the balance change is noticed in the normal type but unnoticed in the child type.

If `BankAccount` were a final class, the inheritance is not possible. Alternatively, if the `addAccount` and `listAccountBalances` methods prevent child type objects and prevent polymorphism, the false reporting is not possible. Final classes preventing unwanted inheritance is the recommended approach.

**Chapter 3: Exception Handling**

In Java and in many other languages, runtime exceptions must either be thrown (or passed back) to the calling function, or instead handled where the exception occurs. The latter is typically the preferred solution, as the former assumes that the calling function will properly interpret and handle the exception generated. While this can be a valid
assumption, consider the example in Figure 5, which illustrates some potential trouble caused by this scenario.

The first part of this example shows the main method intercepting a thrown exception from the doSomethingDangerous method, which attempts to open a non-existent file, generating a FileNotFoundException. But the main method is listening for any type of exception, and prints out a misleading result despite the critical error.

The second part of the example where the same exception is handled where it is generated, is in the doSomethingLessDangerous method. Even with the same attempt to suppress in the main method, the error message is still reported to standard output because it is handled on the spot.

Security Implication: These situations could be exploited to suppress important error messages. It is important to make sure that critical errors are handled in the most appropriate way to the particular software being developed—whether that happens to be with code written by others or in the code being created. In some cases, however, there is no better choice but to throw the handling of exceptions to the calling function.

Chapter 4: File I/O

The code in Figures 6 demonstrates an insecure way to read data from a text file in an expected format. If expectations regarding the format and types of data are not checked, it becomes possible for malicious data to cause undesired behavior. This example reads in student data from a text file which is expected to have two data per line—a student ID number followed by a GPA. A simple Student class was created to maintain the ID and GPA and to implement a toString method. It first attempts to open a file named by the String parameter, which could have come from anywhere. In this
example it is read in the main method from standard input and could be malformed or even malicious. In other contexts, this scenario can be used for query injection attacks.

Security Implications: If anything is wrong when opening the file, a FileNotFoundException will be generated, but the method throws this exception to the calling method rather than handling it gracefully. This can lead to DoS attacks if the exception is repeatedly triggered. Next, this method presumes the integrity of the data in the file, which is assumed to have int-double pairs on each line representing student ID numbers and corresponding GPAs. If any non-numerical or malformed data is present in the file, a NumberFormatException will be triggered. This can lead to Denial-of-Service (DoS) or injection attacks.

Chapter 5: Recursion

Example 1: Recursion is an elegant programming technique whereby a method or function calls itself as part of solving a problem. The key factor is to ensure that recursive calls are made with a smaller instance of the problem, and that a recursive function is properly designed by having a base case to end a sequence of recursive calls. If a base case is omitted or malformed, an infinite recursive call chain will be activated. While an infinite loop will often cause the program to hang while a loop counter races, infinite recursion will often crash the program due to a memory error. Consider the example in Figure 7 which prints out the digits of a number in reverse order to standard output. The algorithm is straightforward, with the modulus operation printing the least-significant digit and the integer division recursively calling with the remaining significant digits. However, no base case is provided to stop the recursion, which will print zero until the runtime stack overflows once all the digits have been divided out. The method body should be wrapped in an if-statement testing for values of n which are strictly greater than zero.

Security Implication: This type of crash, rather than a program stuck in execution, can be exploited for malicious purposes. If an attacker knows about a poorly formed recursive solution, it can be exploited for a DoS attack.

Example 2: Infinite recursion is not the only exploitable scenario involving recursion. If a recursive solution is not applicable to all problem sizes, or if complex problem sizes are not detected, system resources can be wasted by an attacker by providing a valid but too-large problem size. A classic example of this is computing the nth Fibonacci number, which comes with a classic recurrence definition which is easy to code as a recursive program, and is shown in Figure 8. While this code runs fine for smaller problem sizes, it quickly gets stuck for moderately large problem sizes due to its wasteful complexity. Since it re-computes smaller sub-problems, the runtime compounds with exponential complexity as the problem size increases.
**Security Implication:** If an attacker knows a recursive solution has been implemented with the assumption of a small or reasonable problem size, the attacker can attempt to provide a too-large problem size to monopolize system resources and create a DoS attack.

**CONCLUSIONS**

Due to the increasing demand for a more cybersecurity-aware workforce, recent models for cybersecurity education design cybersecurity courses for junior- or senior-level students and offer them as electives, or through a minor program or a concentration. However, this model suffers from the drawback that students might graduate with computer science degrees without having knowledge in cybersecurity. As a result, they are more likely to produce software replete with security vulnerabilities due to poor programming practices. In our proposed model of cybersecurity education, cybersecurity concepts will be embedded throughout the computer science curriculum. In this paper, we demonstrate how to embed key concepts of cybersecurity in the second programming course (CS2).

**REFERENCES**


