Motivating Secure Coding Practices in a Freshman-Level Programming Course

Bryson R. Payne , Aaron R. Walker
Computer Science and Information Systems
University of North Georgia
82 College Circle. USA
Dahlonega, GA 30597
{bryson.payne, aaron.walker}@ung.edu

ABSTRACT: *Secure application development is becoming even more critical as the impact of insecure code becomes deeper and more pervasive in our personal and professional lives. The approach described in this paper seeks to motivate computer science students to write secure code almost from the very beginning by focusing on concrete examples of common software vulnerabilities in the second freshman-level programming course. Sample exercises and assignments are given as examples that can be reused in similar courses. While long-term data collection is still ongoing, initial results are promising enough that the method is presented here in detail to support university faculty interested in incorporating lessons and real-world examples in secure app development in their programming courses at any level.*

Keywords: Secure Application Development, Secure Coding, Security-aware Programming, Application Security

Received: 2 July 2014, Revised 10 August 2014, Accepted 18 August 2014

1.Introduction

As apps and other computer code touch more and more areas of our lives, from smart phones and business applications to medical devices and surveillance systems, from self-driving cars and virtual assistants to drones and robots, secure coding practices are more vital than ever. This paper presents an approach to motivate secure coding practices beginning as early as the second freshman-level computer science programming course by giving students hands-on examples of common software vulnerabilities and developing the error-handling and secure coding tools to support safe program operation, and safe program termination, as appropriate, in a given real-world situation.

The methods and examples described in this work can be reused and modified to fit a variety of levels of programming courses in a variety of languages. The sample exercises and assignments provided in this paper were developed in a Java II course, the second freshman-level programming course at the authors' institution.

2. Background

A great deal of research and pedagogy in secure application development has been aimed at web application development (Noureddine & Domodaran, 2008), including agile development for the web (Ge, Paige, Polak, Chivers & Brooke, 2006).

Works like these establish and advocate for guidelines and best practices in secure coding in web applications for a number of reasons. Web-based software vulnerabilities are fairly prominent, receive significant media attention, and perhaps affect our daily lives in a more pervasive manner, as we make online purchases, pay bills online, and interact and transact over the Internet on a daily basis.

In examining the effectiveness of code review after-the-fact, Edmundson et al. (2013) highlighted the fact that not even a single one out of 30 experienced developers in a study could find all the confirmed vulnerabilities in a previously-written, small web application. The application in question had only seven known vulnerabilities, of just three types, but none of the 30 developers was able to find all seven when conducting a security code review after the application had been written and deployed, underscoring the need for secure application development practices from the very beginning.

For this reason, some attention has been given to developing secure coding practices even in introductory computer science courses. Markham (2009) advocates incorporating security principles in introductory CS1 classes and reports higher engagement and motivation in her students, from using interesting current security topics for classroom discussion to writing secret-message-passing programs. The work presented in this paper extends Markham's work to include motivating examples of real-world code vulnerabilities at the CS2 level, when students have a bit more programming experience and can apply secure coding practices more ably.

Chi, Jones and Brown (2013) developed online modules to teach secure coding principles to a variety of STEM majors, with different examples tailored individually by major, from science and math to engineering and CIS. Using a static analysis tool along with video demos and quizzes to introduce students to code vulnerabilities such as buffer overflows, input validation, file operations and more, students showed significant improvement in perceived familiarity with secure programming after following the modules online. The authors reemphasize that secure coding principles should be taught early and continuously throughout courses involving code development.

It is for good reason that the focus on secure coding has moved earlier in the education of future programmers – as Futcher and von Solms (2008) perceptively note, "*software developers generally ignore the idea of security*" (p.56). Although this is a generalization, accrediting and professional standards organizations have recognized the issue and have provided curriculum guidelines (Cooper et al., 2010) to assist colleges and universities in building CS and related curricula that include information security components. In many institutions, though, security is covered only in conjunction with ethics as a required curriculum component (Markham, 2009), with upper-level electives in security rather than security woven throughout lower-level and upper-level content courses. The work presented in this paper aims to serve as an approach toward incorporating security in application development courses at any level, beginning as soon as the first or second programming course.

The focus on motivation is intentional, as prior work (Kanno, Terada, Yajima, Kamamura & Doi, 2009) emphasized the importance of motivational factors in information security implementation. Kanno's team examined various motivating factors, or drives, from risk management and internal control to requirements from business partners and social responsibility, and also separated motivating factors by whether respondents had experienced information security incidents directly. IT professionals attending security conferences showed different levels of various motivating factors for information security adoption and implementation if they had experienced an information security breach or incident directly, showing higher concern for internal controls, while respondents who had not experienced an information security incident directly reported a greater concern if security were a requirement from business partners, for example.

In the introductory non-major CS/IT course at our institution, as well as in the required Computer Ethics course for CS and CIS majors, motivating examples like those detailed by Newman (2006), including cybercrime, identity theft, and other security threats and vulnerabilities, are used to enhance the relevance of class discussions and deliverables. The work presented in this paper is the result of a first attempt at providing real-world, motivational, and accessible examples of various code vulnerabilities and the secure coding practices that can address those vulnerabilities at the level of a second-semester freshman CS or CIS major.

3. Approach

3.1 Class Discussions

Similar to previous research (Markham, 2009; Newman, 2006), discussions in any computing course can benefit from injecting

topics of current interest, from web security breaches to hacked Twitter accounts, and so on. In the authors' experience, simply taking a small amount of time to acknowledge security incidents as they arise, and where possible, describing possible approaches to securing IT resources against such incidents, can contribute significantly to interest and motivation in a course, including programming courses. The fact that this interest and motivation is centered on information security may be a happy bonus and contribute to greater security focus later in the students' studies.

3.2 First Hands-on Experiences

The importance of secure coding can begin as early as the first input validation, and certainly by the second programming course in the major students should be exposed to bad input examples and write code to handle erroneous inputs. It is easy in the rush of teaching multiple topics, though, to skip over a motivating example as simple as entering a string when an integer is expected as a benign example, or attempting code injection as a malicious intruder might.

Showing the effects of bad user input in even a simple program, like a phone-book/contact-list app or the calculator app developed in Section 3.3 below, can help frame and demystify the input checking and validation that we teach.

If a programming course includes any treatment of database interaction, a straightforward SQL code injection example can open students' eyes to the challenge and complexity of securing both web and traditional apps. A personal favorite SQL code injection example of the authors' can be found in (Munroe, 2007).

In courses that deal with file input and output, a motivating exercise on exception handling can include opening a file on a USB flash drive in any popular application (an Office app, Notepad++, any app will serve just fine), then removing the USB drive at various stages – right before saving, for example –attempting to delete the file at the command line or shell while editing can be instructive, as well. (Of course, some caution is due here, and discussing the possible negative impacts of yanking a USB drive out while in use is both appropriate and fortuitous at this point.) Examining the temporary "*owner*" file left by Microsoft Word in a text or binary/hex editor when a student fails to close a file correctly can be an interesting detour in itself.

After trying other applications to see how they handle (or sometimes fail to handle) missing files and other problems, students can more readily envision the types of exceptional situations that could occur in the regular, or irregular, use of the application they're building.

3.3 A Buffer Overflow Example

Other languages may be better suited to demonstrate the impact of a buffer overflow, C and C++ for example, as these languages typically do not protect against accessing or overwriting data in any part of memory, including checking to see that data written to the default buffer type, an array, is within the boundaries of the buffer/array. However, a motivating example and related discussion around buffer overflows can still be valuable, even in a Java course.

From an information security perspective, buffer overflow vulnerability is one of the most important software code concerns. Buffer overflow occurs when an object, such as an array, has data written to it which exceeds the object's defined bounds. This can lead to a software crash, memory leak, data corruption, or even the execution of malware. Fortunately for us Java is resistant to buffer overflows due to a built-in check for *A*rrayIndexOutOfBoundsException, making examples of such attacks easier to show with examples without exposing a system to risk.

```
import java.util.Arrays;
public class BufferOverflow
{
  public static void main (String [] args)
  {
    int[] lucky = new int[12];
    for (int i = 0; i < 21; i++)
      lucky[i] = (int) (Math.random() * 30 + 1);
   System.out.println ("Your lucky numbers are " + Arrays.toString (lucky) + ".");
  }
}
```

The above program is intended to provide the user with an array of 12 lucky numbers chosen at random between the values of 1 and 30. In order to obtain these random numbers and populate them into the array, a *for* loop is used. Unfortunately, the programmer made a mistake. The array is of length 12 and the for loop allows 21 iterations, a typographical error, perhaps. This, of course, exceeds the bounds of the array and generates an ArrayIndexOutOfBoundsException, forcing a program termination. While Java does not allow access to memory out-of-bounds, the program terminates and provides an opportunity to discuss the impact of various programs failing in this way. In the lucky number app, the impact of unexpected termination is quite small, but in a medical device or self-driving car, for example, a program crash could be much more serious.

Not every programming language is as vigilant as Java, however. Many other languages, like C and C++ mentioned above, do not have built-in protection against buffer overflows and therefore must rely upon the operating system to handle these events (Microsoft Windows uses DEP and ASLR). It is important for beginning programmers to understand the dangers of buffer overflows, especially as they broaden their skills to cover multiple languages.

3.4 An Exception-Handling Example

Whenever input is required from a user, there is an opportunity for errors to happen. If a program prompts a user for a name, but he or she responds instead with a sequence of numbers, this may result in an unexpected or even comical condition when the "*name*" is used elsewhere in the program. If a user is prompted for a number and responds by typing a string of characters instead, however, the program could crash altogether.

Let's consider the following simple calculator program.

```
//*************************************************
// MyCalculator.java
//
// Demonstrates the need for exception handling
//*************************************************
import java.util.Scanner;
public class MyCalculator
{
  //-------------------------
  // Reads user input from the command line to
  // collect two numbers and an operator to
  // perform basic arithmetic functions.
  //-------------------------
  public static void main (String[] args)
  {
    // declare variables to hold
    // the two numbers and operator
    int num1, num2;
    String operation;

    // collect the two numbers from the user
    Scanner input = new Scanner(System.in);
    System.out.println ("Enter the first number:");
    num1 = input.nextInt ();
    System.out.println("Enter the second number:");
```

```
num2 = input.nextInt ();
// collect the operator from the user
    Scanner op = new Scanner(System.in);
    System.out.println("Enter the operation: (+, -, *, /)");
    operation = op.next ();

    // perform the operation
// and output the results
    if (operation.equals ("+")){  // addition
      System.out.println(num1 + " + " + num2 + " = " + (num1 + num2));
    }
    if (operation.equals ("-")){  // subtraction
      System.out.println(num1 + " - " + num2 + " = " + (num1 - num2));
    }
    if (operation.equals ("*")){  // multiplication
      System.out.println(num1 + " * " + num2 + " = " + (num1 * num2));
    }
    if (operation.equals ("/")){  // division
      System.out.println(num1 + " / " + num2 + " = " + (num1 / num2));
    }
  }
}
```

This program prompts the user for two numbers and an arithmetic operator. Both numbers are read as integers and the operator is read as a string. Of note is the fact that there is an assumption that the user will provide the correct data. If a user were to input a letter or a floating point number instead of the integer expected, this program would crash due to an input mismatch exception. Furthermore, this program fails to do anything if the operator input does not match one of the values checked for in the series of conditional statements. This allows not only for the possibility of an error due to a lack of

```
// perform the operation and output the results
// Switch statement has a 'default'
switch (operation){
  case "+":
    System.out.println (num1 + " + " + num2 + " = " + (num1 + num2));
    break;
  case "-":
    System.out.println (num1 + " - " + num2 + " = " + (num1 - num2));
    break;
  case "*":
    System.out.println (num1 + " * " + num2 + " = " + (num1 * num2));
    break;
  case "/":
    System.out.println (num1 + " / " + num2 + " = " + (num1 / num2));
    break;
  default:
    System.out.println ("Operation '" + operation + "' not recognized.");
}
```

validation, but also for logical errors that decrease the value of the program for the end user.

A small enhancement to this program might be achieved by changing the *if* conditionals into a more efficient switch statement.

In addition to cleaning the code up a bit, the switch statement allows for a default case which can be used to handle any input given which does not match the mathematical operators this program is meant to handle. For instance, if the program receives the string "*add*" instead of a "+" character, it will respond with the phrase "*Operation 'add' not recognized*" instead of simply closing.

With the above example, we run into another potential issue. The programmer decided to store numerical values as integers for evaluation. Perhaps it simply was not considered that a user may wish to input a decimal value, or that a decimal value might result from the valid operations. For instance, if a user attempts to divide *3* by *4*, the result provided here will be *0*. Changing the expected value from an integer (int) to a float will allow for the correct calculation – this is an example of how a simple oversight could cause significant risk. If this program were used by an accountant, and all decimal values were treated as 0, the results could be disastrous.

While considering division, it is clear that this program also overlooks division-by-zero errors. Such a condition presents an exception, and abnormal program termination. There are several options available within Java to handle exceptions like division by zero.

Let's examine a new version that uses the try-catch statements to handle exceptions like these:

```
import java.util.Scanner;
public class MyCalculator3
{
  //————————————————————
  // Reads user input from the command line to
  // collect two numbers and an operator
  // to perform basic arithmetic functions.
  //————————————————————
  public static void main (String[] args){
    // declare variables to hold the two
    // numbers and operator
    float num1, num2;
    String operation;

     // try block houses our calculator code and
    // passes an exception to the catch clause
    // if one occurs
    try{
      // collect the two numbers from the user
     Scanner input = new Scanner(System.in);
      System.out.println("Enter the first number:");
      num1 = input.nextInt();
```

```
System.out.println("Enter the second number:");
      num2 = input.nextInt();
      // collect the operator from the user
      Scanner op = new Scanner(System.in);
      System.out.println("Enter the operation: (+, -, *, /)");
      operation = op.next();

      // perform the operation and
      // output the results
      // Switch statement has a 'default'
      switch(operation){
        case "+":
          System.out.println(num1 + " + " + num2 + " = " + (num1 + num2));
          break;
        case "-":
          System.out.println(num1 + " - " + num2 + " = " + (num1 - num2));
          break;
        case "*":
          System.out.println(num1 + " * " + num2 + " = " + (num1 * num2));
          break;
        case "/":
          System.out.println(num1 + " / " + num2 + " = " + (num1 / num2));
break;
        default:
          System.out.println ("Operation '" + operation + "' not recognized.");
          break;
      }
    }
    // catch clause takes effect if the exception
    // occurred in the try block
    catch(java.util.InputMismatchException e){
      System.out.println ("Input was not a valid number.");
    }
    catch (java.lang.ArithmeticException e){
      System.out.println ("You cannot divide by zero.");
    }
  }
}
```

The try block performs the instructions it contains unless an exception is encountered. Here the program is capturing two exceptions – InputMismatchException and ArithmeticException. The division by zero error is defined by the ArithmeticException and if encountered, this program prints a message to the screen stating that it cannot divide by zero. Similarly,

InputMismatchException helps the program by catching non-numeric input provided by a user. If the first number to be considered by the calculator is "*w*", an exception occurs when this character is parsed as a number. Instead of a software crash, the program will print a message to the screen informing the user of their error and then close gracefully.

It should be noted that while these techniques may seem standard fare for beginning programming, these are common mistakes that are easy to make. As such, any lack of input validation or error handling is going to be the kind of low-hanging fruit a malicious user will attempt to discover and exploit during the course of a sustained attack against a system. By stressing the importance of these defensive programming measures, we can prepare the novice programmer for when they will need to be aware of SQL injection and cross-site scripting vulnerabilities in their future studies and work experience.

3.5 The Importance of Comments

Providing clear comments in the program design and flow of execution is often the last consideration of a novice programmer. It seems like unnecessary work, because it is simple text which does not affect the performance of the program. After all, the primary focus in most courses and real-world projects is placed on producing an executable which, at least, meets the expectations of the given assignment.

Instructors often have the unenviable job of sifting through a student's code to discern the logic employed – a task made much more difficult by those students who fail to employ sufficient comments in their code.

Beyond academic evaluation, a student programmer should be aware of the consequences of a lack of comments. The Javadoc tool is invaluable as a documentation engine, yet it is only as efficient as the code comments and format entered by the programmer. The resultant documentation can be used to accurately convey the logic, form and function of the programmer's code. As the software development life cycle places this code out of the original programmer's hands and into those of others, this becomes a measure of efficiency that has substantial business cost associated with it. From this perspective alone, the return on investment for proper code commenting technique becomes immediately apparent.

Here is an example of Javadoc-style comments in a snippet of code:

```
/**
* Returns a Boolean response to whether a given
* String is a palindrome. The string is stripped
* of whitespace via a regular expression.
* The first if statement checks for a length
* of 0 or 1, which would automatically make it a
* palindrome. The second if statement examines the
* first and last characters of the string.
* This is continued by removing the first and the
* last characters of the string and recursively
* evaluating via a substring. <p>
* A palindrome is a sequence of elements which
* reads the same forward or reversed. An example
* would be "amore roma" or the number "123454321."
* @param    str the string which is the
*   palindrome candidate
* @return   a Boolean true/false answer
*/
public static boolean findPalindrome(String str) {
  str = str.replaceAll("\\s", "");
```

```
if(str.length() == 0 || str.length() == 1)
    return true;
  if(str.charAt(0) == str.charAt(str.length()-1))
    return findPalindrome(str.substring(1, str.length()-1));
  return false;
}
```

The value of this type of commenting can be readily apparent once students use the Javadoc executable (found in the bin folder of Java development kit installations) to generate elegant HTML documentation from the above snippet.

Good comments can employ a variety of styles, but they bring a sense of cohesion to a programming team if the team has a common methodology. The security aspect of information assurance can be influenced by even such a small gesture as this - for the ability to develop, maintain, and release quality code according to a set standard regardless of the environmental conditions can guarantee adherence to secure development practices and the delivery of the expected level of service to the customer.

4. Results

The use of relevant, real-world examples of information security vulnerabilities and exploits in class discussions and exercises has produced enough positive anecdotal feedback in its first offering to warrant further inquiry and research.

Lively and engaged conversation like that encountered when discussing security incidents like major data breaches and hacked social networking accounts honestly almost seems out of place in a computer science programming course the first time it occurs. The positive impact of this engagement and the teachable moments such incidents provided in a single semester of focused effort make it attractive enough to refine further, and to share in this paper for others to duplicate and extend.

Students seemed thrilled to get to try to "*break*" their favorite (or least favorite) programs by intentionally creating error conditions like missing files, flash drives, networks, and so on. The brainstorming that followed, on how they might program security measures into their apps that handled similar situations, was among the most creative of the entire course.

Buffer overflows and array index-out-of-bounds errors appeared to make more sense to students when they actually tried to run snippets of code with intentionally-placed errors themselves, rather than just viewing error messages on a slide or on the instructor's screen. And, exception handling can be daunting work in Java, but students reported being better able to understand and appreciate exception handling after building, breaking, and securing code.

This immersion into secure coding techniques has also made students more aware of current information security issues and interested in knowing how common, every-day applications become susceptible to malicious code exploits. After understanding that poor program design can create headlines, students show greater interest in developing software that is resilient to common attacks.

5. Conclusions and Future Work

The approach presented in this paper has shown positive initial results in improving motivation and engagement in developing secure applications in a second-semester, freshman-level computer science II course in Java. The authors intend to extend this implementation into higher-level programming courses, and many of the examples given here have already been added to a security and ethics course at the sophomore level.

One area of interest for future work is gathering longitudinal data for the students involved in this first implementation group to determine the impact of the secure coding exercises as presented upon their understanding, ability and performance at developing secure code in later courses. The authors also plan to develop specific units for a senior-level software engineering

course that reuse and reinforce some of the same principles, and further augment the examples with application-specific security concerns appropriate to the senior-level course.

References

[1] Noureddine, A. A., Damodaran, M. (2008). Security in web 2.0 application development. *In*: Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services (iiWAS '08), p. 681-685. ACM.

[2] Ge, X., Paige, R. F., Polack, F. A. C., Chivers, H., Brooke, P. J. (2006). Agile development of secure web applications. *In*: Proceedings of the 6th international conference on Web engineering (ICWE '06), p.305-312. ACM.

[3] Edmundson, A., Holtkamp, B., Rivera, E., Finifter, M., Mettler, A., Wagner, D. (2013). An empirical study on the effectiveness of security code review. *In*: Proceedings of the 5th International Symposium on Engineering Secure Software and Systems (ESSoS'13), p. 197-212. Springer-Verlag, Berlin, Heidelberg.

[4] Markham, S. A. (2009). Expanding security awareness in introductory computer science courses. In: Information Security Curriculum Development Concerence (InfoSecCD '09), p.27-31. ACM, 2009.

[5] Chi, H., Jones, E. L., and Brown, J (2013). Teaching Secure Coding Practices to STEM Students. In: Information Security Curriculum Development Concerence (InfoSecCD '13), p. 42-48. ACM.

[6] Futcher, L., von Solms, R. (2008). Guidelines for secure software development. *In*: Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology (SAICSIT '08), p.56-65. ACM.

[7] Cooper, S., Nickell, C., Piotrowski, V., Oldfield, B., Abdallah, A., Bishop, M., Caelli, B., Dark, M., Hawthorne, E. K., Hoffman, L. Pérez, L. C., Pfleeger, C., Raines, R., Schou, C., Brynielsson, J. (2010). An exploration of the current state of information assurance education. *ACM SIGCSE Bulletin*, 41 (12) 109-125.

[8] Kanno, Y., Terada, M., Yajima, H., Kamamura, T., Doi, N. (2009). A comparative study on structure of the motivation for information security by security incident experiences. *In:* Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human (ICIS '09), p. 9-16. ACM.

[9] Newman, R. C. (2006). Cybercrime, identity theft, and fraud: practicing safe internet - network security threats and vulnerabilities. In: Information Security Curriculum Development Concerence (InfoSecCD '06), p.68-78. ACM.

[10] Munroe, R. P. (2007). Exploits of a mom. Retrieved June 6, 2014: http://xkcd.com/327/

Author biographies

Dr. Bryson Payne has taught Computer Science and Information Systems at the University of North Georgia for 15 years, and is a Certified Information Systems Security Professional (CISSP®). Dr. Payne served as the inaugural Department Head of CSIS at UNG, and for six years he served as the university's Chief Information Officer, overseeing an IT division with budgets in excess of $4 million annually. He is the author of the book Teach Your Kids to Code (No Starch Press, 2015), holds a Ph.D. in computer science from Georgia State University and has published articles in CIO magazine and numerous scholarly journals.

Aaron Walker is an Information Security Analyst for the University of North Georgia. He also teaches part-time for the Computer Science and Information Systems department. He loves to conduct technical research in malware analysis as well as studying effective means of communicating information security concerns in terms of business impact as well as end user engagement.