Fundamental Defensive Programming Practices with Secure Coding Modules

Jeong Yang, Akhtar Lodgher
Department of Computing and Cyber Security
Texas A&M University–San Antonio
San Antonio, TX, USA
{jeong.yang, akhtar.lodgher}@tamusa.edu

Abstract—**While many vulnerabilities are often related to computing and network systems, there has been a growing number of vulnerabilities and attacks in software systems. They are generally caused by careless software design and implementations, and not putting sufficient effort into eliminating defects and flaws in the software itself. When it comes to building reliable and secure software, it is critical that security must be considered throughout the software development process. This paper presents a series of modules that are designed to introduce security concepts in beginners programming courses. The modules have been developed to teach the fundamental concepts of defensive programming from the freshman year, to ensure that the programming concepts are taught to beginning programmers from a security perspective. These modules are intended to build a strong cybersecurity foundation, which will then be enhanced further in the advanced courses, such as Secure Applications Programming and Secure Software Engineering courses. Both instructors and students can practice defensive programming with these modules in their classroom. The study plans to evaluate the teaching effectiveness of the modules associated with the Model-Eliciting Activity (MEA), an evidence-based teaching and learning methodology.**

Keywords—secure coding, integer error, buffer overflow, input validation, argument validation, defensive programming

I. INTRODUCTION

This paper presents a series of modules that are designed to introduce security concepts in fundamental programming courses. The modules have been developed to teach the fundamental concepts of defensive programming from the freshman year, to ensure that the programming concepts are taught to beginning programmers from a security perspective [1]. They are aligned with the secure coding guidelines and standard rules suggested by the Software Engineering Institute (SEI)'s CERT program [4, 5]. The modules are also aligned with the essential areas of Information Assurance and Security (IAS). The IAS is one of the 18 core Knowledge Areas (KAs) that correspond to topic areas of study in computing provided by the 2013 curricula guidelines for undergraduate degree programs in CS [3]. The modules are under the NICE NCWF curricula categories of Cyber Threats and Vulnerabilities [2]. Each module contains a detailed explanation of the concept, step-by-step instructions of code development, examples of noncompliant code on how to convert them into compliant solutions, and exercise questions to assess student learning.

This work focuses on eight secure coding modules that can be integrated into undergraduate CS1 and CS2 courses. The goal of the modules is to teach the fundamental concepts of the Information Assurance and Security (IAS) / Defensive Programming from the freshman year, to ensure that the programming concepts are taught to beginning programmers from a security perspective. The modules are intended to build a strong cybersecurity foundation, which will then be enhanced further in the advanced courses such as Secure Application Programming and Secure Software Engineering courses. Both instructors and students can practice fundamental defensive programming with these modules in their classroom. With experiences of defensive and secure programming from earlier courses, students can practice secure software engineering with these modules in their junior or senior level courses.

II. SECURE CODING MODULES

TABLE 1 lists secure coding modules along with their student learning outcomes and objectives, and hours taken by a student to complete the modules. The first several modules can be incorporated into the CS1 course by practicing defensive programming that constructs reliable code components to protect itself. For example, input data validation, buffer overflow, and dangerous integer errors are explained with specified code examples. Compliant usage of different numeric data types and their operations are demonstrated. Simple cryptography techniques from other modules can also be incorporated to teach the concept of the character data type. Students will be able to build a small code segment to encrypt data using shift ciphers, through these modules. Then to teach the concept of a loop, the ciphers will be hacked by breaking the shift cipher code using a loop structure.

In the next stage, a few modules deal with common object-oriented issues such as privacy, visibility, the dependency of class members and valid use of method arguments among classes. For example, students must ensure that any changes made in a super-class must preserve all the program invariants that its sub-class depends on because failure to preserve

TABLE I. STUDENT LEARNING OUTCOMES AND OBJECTIVES

	Module	*Student Learning Outcomes / Objectives*	*Hours*	*Security Topic*
Defensive Programming	A	Detect Integer Errors Caused by Improper Use of Arithmetic Operations	1	Integer/Rounding Errors
	B	Demonstrate Data Boundaries to Prevent Buffer Overflows	2	Buffer Overflow
	C	Validate Input Data by Checking Type, Length, Range	2	Input Validation
	D	Safely Use Numeric Types and Their Operations	2	Safe Numeric Operations
	E	Encrypt and Decrypt Text using Cipher	4	Encryption/Decryption
Object-Oriented Issues	F	Use Members of a Class Properly to Control Access	5	Member Accessibility
	G	Validate Arguments in Method Operations	3	Argument Validation
	H	Catch Unexpected Behaviors for Sensitive Information	3	Exceptions

dependencies can cause security vulnerabilities. Data members of this class will be exposed by declaring them as public or protected are prone to unexpected attacks. The vulnerability of such attacks can be reduced by increasing the privacy level to private declarations.

A. Detecting Integer Errors Caused by Improper Use of Arithmatic Operations

This module is to use integer data types and their operations in a safe manner to avoid improper use of arithmetic operations. Upon completion of this module lesson, students will be able to demonstrate ways to prevent loss of precision when converting primitive integers to floating-point values and detect integer errors and convert integers to floating-point values for floating-point operations.

Integer Errors: When using integer arithmetic to calculate a value for assignment to a floating-point variable, improper use can lead to a loss of information. Integer arithmetic produces integral results, discarding any possible fractional remainder. Furthermore, there can be a loss of precision when converting integers to floating-point values. When used improperly, the results of integer arithmetic will be inaccurate, either by a small amount or in the case of the overflow, to a significant degree. The code block is an example of improper usage of integer arithmetic.

```
short a = 28901;    int b = 1097;
long c = 1802194120923171293L;

float d = a / 17;        // d is 76.0 (truncated)
double e = b / 28;       // e is 39.0 (truncated)
double f = c * 4;        // f is -9.1179793305293046E18
                         // because of integer overflow
```

There are two ways to solve this problem. The first solution solves the issue by casting the integers as floating, turning the integer arithmetic to floating arithmetic. For the last issue with the double, c was type-casted as a double as it has enough bits to handle such a large number.

```
short a = 533;    int b = 6789;
long c = 4664382371590123456L;

float d = a / 7.0f;              // d is 76.14286
double e = b / 30.;              // e is 39.1785
double f = (double)c * 2;    // f is 9.328764743180247E18
```

The second solution circumvents type casting by declaring before operations and redefines the integer variables as floats. Then performs arithmetic operations on them, leading to accurate answers.

```
short a = 533;
int b = 6789;
long c = 4664382371590123456L;

float d = a;   float e = b;   float f = c;

float d /= 7;         // d is 76.14286
double e /= 30;    // e is 39.1785
double f *= 2;       // f is 9.328764743180247E18
```

Rounding Errors: When performing mixed operations, one of the possible errors that can occur is a rounding error.

```
int a = 60070;    int b = 57750;
double value = Math.ceil(a/b);
```

Due to the division being an integer division, the inner division result is truncated to 1, and instead of rounding up to 2, the final result is 1. The solution to this is straightforward.

```
double value = Math.ceil(a/((double) b));
```

As a result of the cast, the other variable, a, is automatically promoted to a double, and the result comes out to 2 as intended.

B. Demonstrating Data Boundaries to Prevent Buffer Overflows

This module is aimed to use different integer data types and their operations in a safe manner to avoid buffer overflows. Programs should not allow mathematical operations to exceed the integer ranges provided by their primitive integer data types. Upon completion of this module lesson, students should be able to identify the boundaries of various integer data types and demonstrate how to detect and correct integer buffer overflow.

Integral types in Java have inclusive ranges based on its type and their ranges are not symmetric from negative to positive as shown in TABLE II. A negative number of each minimum value is one more than each positive maximum value. Even unary negation can overflow if that is applied to a minimum value and abs() method can overflow if the given the minimum int or long as arguments. As integer operators don't indicate overflow or underflow, they can result in incorrect computations and unexpected outcomes. Compliant technologies need to prevent these overflows.

TABLE II. INTEGER TYPES AND RANGES IN JAVA

Type	Byte	Bit Representation	Range
byte	1	8-bit signed two's-complement	−128 to 127
short	2	8-bit signed two's-complement	−32,768 to 32,767
char	2	16-bit unsigned integers	\u0000 to \uffff (0 to 65,535)
int	4	32-bit signed two's-complement	−2,147,483,648 to 2,147,483,647
long	8	64-bit signed two's-complement	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

For example, the following code example is susceptible to integer overflow:

```
public byte operations (byte n1, byte n2, byte n3)
    return n1 + (n2 * n3);
```

To make the code more secure to prevent the overflow, safeAdd and safeMultiply functions can be used for necessary precondition checks required for addition and multiplication operations. Each function throws an exception when an integer overflow would otherwise occur; any other conforming error handling is also acceptable.

```
public byte operations(byte n1, byte n2, byte num3)
    return safeAddition(n1, safeMultiplication(n2, n3));
```

C. Validating Input Data by Checking Type, Length, Range

This module is to validate input data by checking their type, length, and range. It focuses on checking exceptional values from floating-point inputs, not using floating-point variables as loop counters, and using conversions of numeric types to narrower types without losing or misinterpreting data. This is to practice defensive programming that constructs reliable code components to protect itself. Conversions of numeric types to narrower types can result in lost or misinterpreted data if the value of the wider type is outside the range of values of the narrower type. Consequently, all narrowing conversions must be guaranteed safe by range-checking the value before conversion. The following is an example of a code where data can be lost or result in misinterpreted data outside the range of the byte type.

```
int value = 128;     func(value);
public void func(int i)
   byte b = (byte) i;  // b has value -128 ( no -128 in byte)
```

This code can be written securely by checking whether the integer value provided is within the range of maximum and minimum values supported by byte.

```
public void func(int i) {
   if ((i < Byte.MIN_VALUE) || (i > Byte.MAX_VALUE)){
        throw new ArithmeticException("Out of range");
   }
   byte b = (byte) i;
}
```

D. Safely Using Numeric Types and Their Operations

This module is to use numeric data types and their operations in a safe manner to avoid unexpected results. One example is to avoid to use bitwise and arithmetic operations on the same data. While bitwise operations do have their base 10 equivalents, their operations should not be used with normal arithmetic operations due to obscuring programmer's intention and reducing readability. When it comes to outputting a grouping of bit integers, and pack them into a single variable, we need to stick to bitwise. In the following code block, the result is meant to hold a bit collection, not a numeric value (arr[] is a byte array initialized to 0xFF):

```
byte[] arr = new byte[] {-1, -1, -1, -1};
int result = 0;
for (int i = 0; i < arr.length; i++) {
     result = ((result << 8) + arr[i]);
}
```

In the bitwise operation, the value of the byte array element arr[i] is promoted to an int by sign-extension. When a byte array element contains a negative value, the sign-extension propagates 1-bits into the upper 24 upper bits of the int. The unexpected behavior might occur if it is assumed the bytes are an unsigned type.

E. Encrypting and Decrypting Using Cipher

This module is for students to have an understanding of basic encryption and decryption using a Caesar cipher. Some other ciphers are also introduced. Caesar's encryption makes messages secret by shifting each letter three letters forward in the alphabet (sending the last three letters of the alphabet to the first three). It process mathematically, first replace each letter by an element of 26, that is, an integer from 0 to 25 equal to one less than its position in the alphabet. In a computational algorithm, it can be represented by the function f (p) = (p + 3) mod 26, where p represents an integer that is associated with an alphabet letter. To decrypt Caesar's cipher, the inverse of f, f^{-1}, is used: $f^{-1}(p) = (p-3) \bmod 26$. Here, each letter is shifted back three letters in the alphabet with the first three letters sent to the last three letters. To enhance security, a generalized cipher can be used using f(p) = (ap+b) mod 26 where a and b are integers. Upon completion of this module, students should be able to understand how the cipher works, how to encode a simple alpha text and decode the encoded text using Ciphers, describe the basic concept of how to break the cipher code, describe the code for hacking the Cipher, and demonstrate the correct execution of the hack using several examples.

F. Using Members of a Class to Control Access

This module is to declare and use members of a class properly to control access. It focuses on limiting accessibility of fields of a class, not exposing private members of an outer class from within a nested class, and not returning references to private mutable class members.

A nested class is a class whose declaration occurs within the body of another class or interface. Its use is error-prone unless the semantics are well understood. One of the features of a nested class is that it has access to the private fields of its outer class, and by consequence, if this nested class is made public, anything accessing said public nested class has access to the same private fields. The following is a code block showing how a public inner class can affect the private fields of an outer class:

```
class Coordinate {
      private int x;       private int y;
      public class Point {
           public void getPoint()
                   System.out.println("(" + x + "," + y + ")");
      }   }
class Main {
      public static void main(String[] args) {
           Coordinate c = new Coordinate();
           Coordinate.Point p = c.new Point();
           p.getPoint();
      }
}
```

Not to expose the private members of an outer class from within a nested class, changing '**public**' of the inner class to '**private**' solves the issue, then the code block will not compile due to Main attempting to access a private nested class outside of its private scope.

G. Validating Arguments in Method Operations

Through this module, students should be able to validate arguments on method operations. Failure to validate method arguments can result in incorrect calculations, runtime exceptions, violation of class invariants, and inconsistent object state. Whenever arguments are passed to a method, programmers run the risk of having an invalid object state of argument during its use as shown below:

```
private Object myObject = null;
void setMyObject(Object object) {
       myObject = object;
 }
 void useMyObject() {
       // Perform some action using the object passed
 }
```

Checking for valid argument before using is integral to programming securely as follows:

```
private Object myObject = null;
void setMyObject(Object object) {
       if (object == null) {
          // Handle null state
       }
       if (isObjectStateInvalid(object)) {
          // Handle invalid state
       }
       myObject = object;
   }
   void useMyObject () {
       if (myObject == null) {
          // Handle null condition
       }
       // Perform some action using the object passed
   }
```

H. Catching Unexpected Behaviors for Sensitive Information

The risk is that filtering failure of sensitive information when propagating exceptions often results in leaks that can help attackers develop further exploits. They may change input arguments to expose the internal structures of the application. For example, a FileNotFoundException message can reveal information about the file system layout, and its exception type reveals the absence of a requested file.

```
try {

   FileInputStream fis =
  new FileInputStream(System.getenv("APPDATA") + args[0]);
} catch (FileNotFoundException e) {
        throw new IOException("Unable to retrieve file", e);
    }
```

Programs must filter both exception messages and their types not to expose sensitive information. Thus, the above issue can be fixed by only permitting files to be opened by the user as follows:

```
FileInputStream fis = null;
try {
  switch(Integer.valueOf(args[0])) {
    case 1:
        fis = new FileInputStream("c:\\home\\file1");
        break;
    // More options
    default:
        System.out.println("Invalid option!");
        break;
    }
 } catch (Throwable t) {
        MyExceptionHandler.report(t);
 }
```

III. Evaluation Plan

The study plans to evaluate the teaching effectiveness of the secure modules associated with the Model-Electing Activities (MEAs) [8, 9], a proven effective teaching/learning method to help engineering students become better problem solvers. The results of student learning will be investigated to improve the design of the intervention by using the design experiment methodology [10]. This methodology allows the investigation of how a particular intervention affects student learning and instructor teaching practices [11].

The instructor effectiveness and students' attitudes and interest will be studied. The study of instructor effectiveness will be guided by the questions: a) Can MEAs help instructors implement cyber security modules in classroom teaching and how they utilize the MEAs in teaching practices? b) Can the implementation of cyber security modules through MEAs help researchers and instructors assess students' problem-solving strategies and their learning and performance? c) To what extent do instructors change their attitudes towards student learning and their teaching practices because of the implementation of cyber security modules through MEAs?

The study of students' attitudes and interest will be guided by the questions: a) Can the implementation of cyber security modules through MEAs change students' attitudes towards learning in computer science? b) Can the implementation of cyber security modules through MEAs enhance students' interest, willingness, and confidence in computer science?

IV. Conclusion and Future Work

This paper presents eight secure coding modules that are designed to be integrated into undergraduate fundamental programming courses. The goal of the modules is to teach the fundamental concepts of the Security and Defensive Programming from the freshman year, to ensure that programming concepts are taught to beginning programmers from a security perspective. The modules are currently available through the NSA public library for use: CLARK Cybersecurity Library [7]. As part of the grant project [6], these modules will be taught in CS1 and CS2 courses at three institutions (Texas A&M University-San Antonio, Laredo College, San Antonio College) with the MEAs incorporated as a strategic approach to teach the concepts of the security modules.

Acknowledgement

Partial support for this work was provided by the National Science Foundation (NSF)'s grant project entitled "Recruiting and Retaining Students into Computing" under the award #1832433.

References

[1] Akhtar Lodgher and Jeong Yang, "Cyber Security Modules for Core, Major and Elective Courses in the Bachelor of Science (BS) Computer Science Curriculum," National Security Agency (NSA) Grant, Sep 2017-Aug 2018.

[2] William Newhouse, Stephanie Keith, Benjamin Scribner, Greg Witte, "National Initiative for Cybersecurity Education (NICE) Cybersecurity Workforce Framework," National Institute of Standards and Technology (NIST), U.S. Department of Commerce, 2017, https://doi.org/10.6028/NIST.SP.800-181.

[3] The Joint Task Force on Computing Curricula (2013) Association for Computing Machinery (ACM) and IEEE-Computer Society, Computer Science Curricula 2013 Curriculum Guidelines for Undergraduate Degree Programs in Computer Science, Dec 2013.

[4] Software Engineering Institute (SEI) at Carnegie Mellon University. Curricula: Software assurance Materials and Artifacts. Retrieved from https://www.sei.cmu.edu/education-outreach/curricula/software-assurance/.

[5] Software Engineering Institute (SEI) at Carnegie Mellon University. Secure Coding Standards. Retrieved from https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards.

[6] Jeong Yang, Akhtar Lodgher, and Young rae Kim, "Recruiting and Retaining Students into Computing," National Science Foundation (NSF) Grant, Oct 2018-Sep 2021.

[7] CLARK Cybersecurity Library, Retrived from https://clark.center/home.

[8] Moore, T.J. (2008). "Model-Eliciting Activities: A case-based approach for getting students interested in material science and engineering," Journal of Materials Education, v.30 (5-6), pp. 295-310.

[9] Moore, T. J., Miller, R. L., Lesh, R. A., Stohlmann, M. S., & Kim, Y. R. (2013). Modeling in engineering: The role of representational fluency in students' conceptual understanding. Journal of Engineering Education, 102(1), 141-178.

[10] Allan Collins, Diana Joseph, Katerine Bielaczyc, Design Research: Theoretical and Methodological Issues, The Journal of the Learning Science, 13(1), 15-42, 2004.

[11] Nuñez, A.-M. (2015). "Hispanic-Serving Institutions: Where are they now?" A commissioned paper presented at the meeting "Hispanic-Serving Institutions in the 21st century: A convening" at the University of Texas El Paso. El Paso, TX.