Teaching Secure Program Design

Nadia Jones, Qingrui Yu, Karen Schell and Huiming Yu

***Abstract*-Developing secure software applications is becoming very critical because many different types of attacks are caused by software vulnerabilities. In order to effectively teach secure software engineering, we have developed a course module titled "Introduction to Secure Program Design". This paper presents the content of this module and reports our teaching experiences. This module was successfully taught in the COMP 280 Data Structures class during the Fall 2018 semester in the Department of Computer Science at North Carolina A&T State University. Our experience exhibited that teaching this module helped students not only gain knowledge and understanding about the impacts of input flaws and buffer overflows, but also they gained significant knowledge about the practice of designing secure programs. Students' surveyed responses and feedback reflected that this module was very valuable towards their educational experience. This content could be taught in second year sophomore classes of software engineering, computer science and information technology.**

***Index Terms*-secure program design, input validation flaws, buffer overflow**

I. INTRODUCTION

Computer programming design is defined as the architecture and documentation of procedures used in developing software. In the Software Development Life Cycle, the design phase is the segment of development where the resources needed for hardware and software are recognized and the logical methods that will be used are determined. Software should always be designed from the very beginning, with security in mind. An ideal application will always be designed from the top down and contain secure practices throughout the entire development. A good example of a secure design practice includes that everyone is thoroughly familiar with the design, its ins and outs, and captures other possible threats identified by others. When designing software, developers should keep in mind the idea of "least privilege", meaning that every access point should only allow the minimum access needed to accomplish its necessary functions [9].

Manuscript received March 1, 2019; revised April 1, 2019. This work was partially supported by National Science Foundation under the award number 1662469.

H. Yu is with the Department of Computer Science, North Carolina A&T State University, Greensboro, NC 27411 USA (e-mail: cshmyu@ncat.edu).

N. Jones was a MS student of the Department of Computer Science, North Carolina A&T State University, Greensboro, NC 27411.

Q. Yu was a MS student of the Department of Computer Science, North Carolina A&T State University, Greensboro, NC 27411 USA (e-mail: yuqinrui@outlook.com).

K. Schell is with the Department of Computer Science, North Carolina A&T State University, Greensboro, NC 27411 USA (e-mail: klshnell@ncat.edu).

There are several reasons why programs should be designed with security in mind, especially when major companies provide services to key customers. Companies have information assets that they need to keep protected. Obviously, it is in the best interest of companies to keep these assets safe from threats in order to avoid potential financial loss, trade secrets from being stolen, possible loss of customers, and gaining an overall negative reputation. In order to decrease the harm done to a company, they should put forth special effort to help their developers protect and secure these assets.

The purpose of a design program is to solve given problems. However, designing without keeping security in mind will lead to severe problems and may result in unexpected consequences. Since malicious software can attack unsecure programs and get information without permission, computer science students should always keep security in mind when designing and implementing software to anticipate all possible threats and prevent attacks [1, 2].

Education is a very powerful tool in promoting secure program design. By understanding software vulnerabilities and possible threats, developers can save valuable time and create secure applications while users can be better assured that their information is safe. Most Computer Science, Information Management System, and Software Engineering curriculums do not include secure software engineering content. The students are not adequately prepared to handle this expectation. Based on this demand, the course module entitled "Introduction to Secure Program Design" has been developed to help students learn secure software engineering related practices and understand issues. The module covers computer security concepts; input validation flaws and buffer overflows; how using a secure program design assists with preventing input flaws and buffer overflow; and a laboratory exercise.

In this paper, we present a new course module entitled "Introduction to Secure Program Design", and discuss the teaching experience of implementing the program. In the second section, the teaching objective is discussed. The Data Structures course is described in section 3. The details of the Introduction to Secure Program Design module is presented in section 4. In section 5, the results of the teaching experiment are discussed. The conclusion is presented in section 6.

II. TEACHING OBJECTIVE

Designing and implementing secure software applications is becoming very critical in today's society. It results many new requirements for software developers. In order to prepare future students for the expectations of the Cybersecurity workforce and for them to effectively learn about secure software engineering, we have developed an "Introduction to Secure Program Design" module for the Introduction to Design and Data Structure computer science classes. The

ISBN: 978-988-14048-6-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

objective of this module is to enhance software engineering, computer science and information management system curriculums by providing knowledge of input validation flaws, buffer overflows and technology to prevent them. Instructors can teach this module in computer science, information technology, and software engineering sophomore classes.

Upon completion of this module, students should be able to 1) understand why secure program design is very important, 2) gain significant knowledge of secure program design, input validation flaws, buffer overflow, how to prevent input flaws and buffer overflow vulnerabilities, and 3) apply the acquired knowledge to real world applications.

III. COMP 280 DATA STRUCTURES

Comp 280 Data Structures is a required course for undergraduate students who's major is computer science. This course takes students that have completed, as a prerequisite, a foundational programming class to the next level of learning data structures. This course makes use of abstractions (algorithms, data types) and programming structures (pointers, dynamic memory, and linked data structures). The course examines essential data structures (stacks, queues, trees, linked lists, and graphs). It analyzes and implements techniques such as hashing, sorting, searching, and priority queues to solve general problems. The emphasis of the course is on building computer programs that implement essential data structures. The students journey through the thought process of programming efficiency and effectiveness in order to handle problems like the speed of data retrieval, storage and management. Further, they gain experience with performing the presentation and discussion of their design logic; and handle questions and feedback from their peers. In order to enhance a student's cybersecurity knowledge, we taught the Introduction to Secure Program Design module in COMP 280 Data Structure class.

IV. COURSE MODULE: INTRODUCTION TO SECURE PROGRAM DESIGN

The Introduction to Secure Program Design module consists of five parts that include Introduction, Secure Program Design Considerations, Insecure Programs, Safe Program Design and a Laboratory Exercise.

A. Introduction

In the introduction section, we briefly describe the importance of secure software engineering, secure program design considerations, ensuring program best practices, and the impact of insecure programs. Several examples are presented to demonstrate that insecure programs can introduce vulnerabilities and increase the possibility of resulting in an attack.

B. Secure Program Design Consideration

Top-level design refers to the steps to organize and develop an application and to identify specific components that ensures that the application perform according to their guidelines or requirements. In most cases, the design phase is based on the requirements of the project and should not only implement these requirements but also ensure that security consideration included during this phase [8]. Developers should set security goals and ensure the priority of these goals throughout the entire design of a program. The earlier that security is taken into consideration in the development process, the fewer errors that may be exposed later. Security measures are often ignored in the design phase and often leads to problems. Typically a designer may assume that many security flaws occur in the implementation and testing phases of software, but often these results actually occur mostly in the design phase. Some areas of a program where users receive direct exposure (such as user authentication) need to be designed before the implementation phase of the development cycle. During implementation, developers are only concerned with implementing code based on the design requirements and not security requirements. This is why it is important for software to be designed with security considerations. The design phase should outline how to implement the functionality of features, while ensuring that these features are secure. Secure features have a "functionality that is well engineered with respect to security, such as rigorously validating data before processing it" [10]. It is very important to consider security issues early in development, as opposed to the end, where anyone can make adjustments to software and possibly harm it. Once a user becomes more advanced in developing software, he or she should implement the abstraction and decomposition principles. Abstraction reduces complexities within a system by removing extra details and isolating important elements to easily manage the design [10]. Decomposition refers to the process of breaking down and describing a problem or generalization that make up an entire problem [10].

C. Insecure Programs

Developers failing to consider security when designing a program is a continuous problem for programs and applications. Security is not seen as an important concept until the final phases of the Software Development Life Cycle (SDLC). When designing, developers are not always aware of the possible vulnerabilities and can be oblivious to what they even look like. Some issues that arise with the lack of secure program design include poor use of cryptography, buffer overflows, flawed input validation, and overall weak structural security. In the following sections, buffer overflows and flawed input validation will be discussed, including how the vulnerabilities occur, their impact, and possible solutions.

Input Validation Flaws

Input validation is defined as the validation of any data or information before inputting it into a program or application [5]. Validating input is extremely important for the safety of applications and programs. One of the issues with input validation is that many times an application does not verify the input leading to input errors which provide opportunities for attacks. Thus, input validation flaws can be described as incorrect or inappropriate validation of data from a client or environment.

First, we introduce to the students what is an input validation and what is the impact of input validation flaws. We use an example to show the results of an input validation flaw. We use the Java program shown in figure 1 to demonstrate this concept [x].

A simple validation flaw can easily lead to other serious vulnerabilities such as interpreter injection, Unicode attacks, file system attacks, and buffer overflows [6]. Figure 1 is a

ISBN: 978-988-14048-6-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

simple program that has room for several input errors. A user is prompted to enter a positive integer. The user should check for two input conditions - the input is an integer and that the integer is greater than 0.

Buffer Overflow

A buffer is a block of memory in a program used to temporarily store data [4]. A buffer serves multiple purposes such as holding data while input or output information is being transferred and moving data between processes on a computer [4]. An easy way to think of a buffer is a way that a program remembers certain information [1].

```
public class FlawedInput {
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      System.out.println("Enter a positive integer value");
      int x = input.nextInt();
      System.out.println("You have entered an acceptable positive integer value");
      }
}
```

Fig. 1. Input Validation Flaw Example

In this example, if a user enters a value such as “seven”, the errors shown in Figure 2 will be displayed. The java code catches this error as an Input Mismatch Exception because the user provided characters, rather than an integer. Fortunately, java is a language that automatically throws exceptions, but users should not rely on this, because not all programming languages have this benefit. If a user enters an integer value, such as -7, the program will run successfully and display “You have entered an acceptable positive integer value”, although the value that the user entered was not a positive one. If user inputs are not properly validated and an attacker gets into a system, they have the ability to harm the program application.

```
Exception in thread "main" java.util.InputMismatchException
   at java.util.Scanner.throwFor(Scanner.java:840)
   at java.util.Scanner.next(Scanner.java:1461)
   at java.util.Scanner.nextInt(Scanner.java:2091)
   at java.util.Scanner.nextInt(Scanner.java:2050)
   at flawedinput.FlawedInput.main(FlawedInput.java:17)
```

Fig. 2. Input Validation Flaw Output

A buffer overflow is a serious security vulnerability in software and continues to be common in older, as well as, newer applications [2]. A buffer overflow occurs when a program tries to put more data in a buffer than what is allocated. Buffer overflows, also occur, when a program tries to write outside of the boundaries of a designated block of memory [2]. The most common type of buffer overflow is one in which an attacker attempts to overload a stack so that malicious code can be implemented [9].

Although buffer overflows are difficult to discover and exploit, attackers are still able to find these vulnerabilities, and use them as opportunities to inject malicious data into programs [2]. Once a buffer overflow occurs, an attacker can overwrite data that controls program execution and use this to conduct further attacks on the program [3]. Buffer overflows can be found in all types of products or applications, and typically occur within code that is too complex for a programmer to understand and predict its behavior [2].

There are several examples of buffer overflow attacks. A typical example would be an attacker sending data to a program and storing it in an undersized stack buffer. When a program call is made to this stack, the data, including the function’s return pointer, will be overwritten and the value of the return pointer is transferred to malicious code originating from the attacker’s data [2]. Writing data outside of the given memory space can cause the corruption of data, infinite loops, program crashes, and further exploitation of code [2]. Other impacts of buffer overflow attacks include improper programming behavior, inability to access memory, programs producing incorrect results, and easy access to elements that can allow a user to alter an application’s security [1].

Consider the example in Figure 3, where a simple Array is defined to hold three integers. The buffer overflow occurs where i=3, and a user tries to access an address space that is not available.

```
public class BuffOver {
   public static void main(String Args[]){
      int[] simpleArray = new int[3];
      for(int i=0;i<4;++i){
      simpleArray[i] = i;
      }
   System.out.println(“You have created a secure array!”);
   }
}
```

Fig. 3. Buffer Overflow Example

A simple Array has been created to hold three integer values in memory and the program tries to write data outside of the array’s boundary. This extra data will cause program space to be overwritten and could likely cause data corruption [7]. Data corruption can cause a program to crash, or worse, allow an attacker to execute malicious code, which only leads to other serious problems [8]. No data will be printed to the screen because the program will terminate and a user will receive the error shown in Figure 4. This output is another example of a java exception. The output demonstrates that a user is trying to access a memory space that is outside of the bounds declared for the array.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
      at buffovertest.BuffOver.main(BuffOver.java:18)
```

Fig. 4. Buffer Overflow Error

ISBN: 978-988-14048-6-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

D. Secure Program Design

It is important emphasize if a software engineer wants to develop secure software applications then he/she must implement secure program design from the very beginning of the development cycle. A secure program design practice includes several steps that are extremely important when it comes to ensuring that applications are not vulnerable to attacks.

Data Validation Strategies

To control user input so that a user cannot deliberately enter harmful input into a program, a designer needs to ensure that all inputs are checked and validated properly. Two solutions for ensuring the proper validation of data involve accepting known good input and rejecting known bad input. In Figure 1, the program must ensure that an integer is entered that is greater than 0. If the input does not meet this requirement, the program should not run successfully as it did in the previous example. The modified program is shown in Figure 5. This program uses a try/catch block to ensure that the appropriate value is input into the program. If a user takes the necessary precautions to ensure all inputs are correct, the program should work similar to the output displayed in Figure 6. For this example, the user should keep in mind the principles of accepting known good and rejecting known bad data. In Figure 6, a user checks various types of input to ensure that the requested one is accepted. If it is not accepted, the program notifies the user. The first three attempts show that rather than entering a positive integer value a user has entered a string, a negative number, and a non-integer value. For these three cases, these values are not accepted. This is an example of rejecting known bad. Finally when the user enters the value '9', it is accepted because it is a positive integer value. This is an example of accepting known good. An optional solution to improving data validation flaws is to include integrity checks. Integrity checks should be included anywhere that data passes from a trusted to a less trusted source. An example of this would be transferring data from the application to a user's browser via a hidden field [6].

Buffer Overflow Prevention

To avoid a buffer overflow, a user should always check the array size before writing data to a buffer. If we recall the example demonstrated in Figure 2, we note that a buffer overflow will occur when a user tries to access the fourth address space that was not actually available. There are two ways to correct this problem. The first solution is to increase the size of the array (See Figure 7) and the other would be to access an appropriate address within the array's memory (See Figure 8). In Figure 7, the simple array is redefined so that it can now hold four integers. In Figure 8, the for loop termination condition has changed according to the array size.

```
import java.util.Scanner;
import java.util.InputMismatchException;
public class FlawedInput {
   public static void main(String[] args) {
   Scanner input = new Scanner(System.in);
   int num = 0;
   boolean loop = false;
     while (!loop) {
      try {
      System.out.println("Enter a positive integer value.");
      num = input.nextInt();
      while (num < 0 || ((num < Integer.MIN_VALUE || num >
Integer.MAX_VALUE))) {
      System.out.println("You have not entered an acceptable
positive integer value");
      num = input.nextInt();
      }
     loop = true;
     System.out.println("You have entered an acceptable
positive integer value");
     }
     catch (java.util.InputMismatchException ex)
      {
      String wrongNum = input.nextLine();
      System.out.println("You have not entered an acceptable
positive integer value.");
      }
     }
     }
}
```

Fig. 5. Input Validation Flaw Solution

```
Enter a positive integer value.
nine
You have not entered an acceptable positive integer value.

Enter a positive integer value.
-9
You have not entered an acceptable positive integer value
99999999999999999
You have not entered an acceptable positive integer value.

Enter a positive integer value.
9
You have entered an acceptable positive integer value
BUILD SUCCESSFUL (total time: 18 seconds)
```

Fig. 6. Input Validation Flaw Output

```
public class BuffOver {
   public static void main(String Args[]){
     int[] simpleArray = new int[4];
     for(int i=0;i<simpleArray.length;++i){
            simpleArray[i] = i;
            }
     System.out.println("You have created a secure array!");
     }
}
```

Fig. 7. Buffer Overflow Solution 1

ISBN: 978-988-14048-6-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

```
public class BuffOver {
   public static void main(String Args[]){
     int[] simpleArray = new int[3];
     for(int i=0;i<3;++i){
      simpleArray[i] = i;
     }
    System.out.println("You have created a secure array!");
  }
}
```

Fig. 8. Buffer Overflow Solution 2

In both of these examples, our simple buffer overflow problem has been resolved and the program will successfully run and display the "You have created a secure array!" message. This is a simple example and users should always be cautious of where the boundary is. Java has built in capabilities that check for an array boundary and will display the error message as shown in Figure 4, but other programming languages may not. This can leave room for attacks.

E. Laboratory Exercise

A laboratory exercise was developed to help students understand buffer overflows and input validation flaws. The laboratory consisted of two parts - a Java project assignment, and questions and answers. Students should be able to use this module as a guide to develop a vulnerable program as well as provide an acceptable solution for this program.

Secure Program Design Assignment

We ask students to develop a java program to demonstrate secure program design strategy concept. This program should contain one instance of integer overflow, one instance of a buffer overflow, and one instance of an input validation flaw. Students should exploit the vulnerabilities of the program and print the results. Once the program has been successfully exploited, students will correct these errors, execute the program again, and display the results of the successful program.

Follow Up Questions

We have developed a set of questions to help students learn secure program design. After students study the module and complete the given program assignment, he/she should have capability to answer the following questions.

a) Describe the difference between an input error and input validation flaw.
b) Give one example of a buffer overflow.
c) Answer the question –"What is the relationship between an input validation flaw and a buffer overflow?"
d) Answer the question – "What types of input errors did you use for the assigned program and what were your results?"
e) Give two examples of how to prevent buffer overflows.
f) Give two examples of how to prevent input validation flaws.

V. TEACHING EXPERIMENTAL RESULTS

The Introduction to Secure Program Design module has been successfully taught in COMP 280 Data Structure class in the Department of Computer Science at North Carolina A&T State University in the fall of 2018 and received excellent results. The COMP 280 class had two sessions. All of students were sophomores. To evaluate the students' reactions and obtain feedback about using the module, we conducted a pre-survey before the module was taught and post-survey to evaluate students learning outcomes. Twelve students were in session I and fourteen students were in session II and were given the pre and post survey for the course module. We received excellent results. The student's excitement level for learning secure program design was very high. They showed great eagerness and excitement to learn.

The survey questions were developed with the intent to understand how well the students gained knowledge from the experience. The pre-survey consists of six questions that include extracting 1) knowledge about buffer overflows, 2) knowledge about input validation flaws, 3) knowledge about secure program design practices, 4) understanding the importance of designing secure code, 5) the concept of buffer overflow prevention and 6) input validation flaws prevention. Based on how well the students felt that they understood the material, they would give a score for each item with a scale of 1 (very low), 2 (low), 3 (medium), 4 (high), and 5 (excellent). The post-survey included six questions in an attempt to understand how well the students understood the material after the instructor explained the module. The post-survey questions are same as pre-survey. The survey results are shown in fig.9 and fig.10. The blue bar is the pre-survey results. The red bar is the post-survey results. The findings illustrate how much the students acknowledged that they had improved on their understanding of the material presented in the different areas of the module.

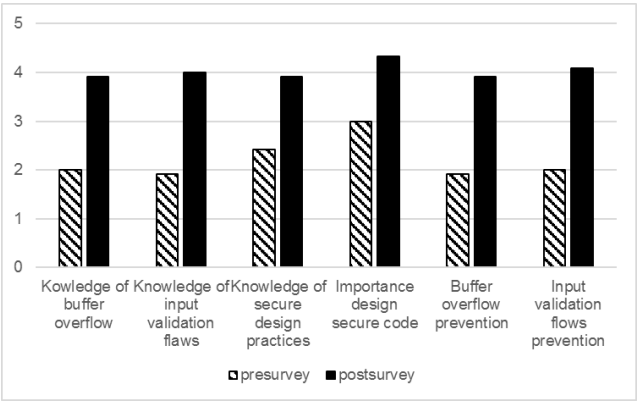

Fig. 9. Students Level of Knowledge Survey of Session 1

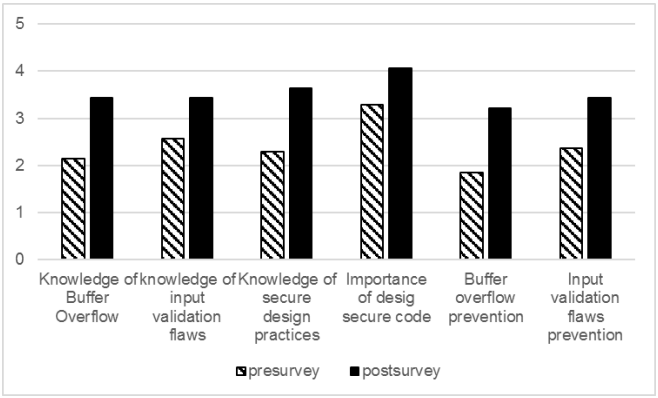

Fig. 10. Students Level of Knowledge Survey of Session 2

ISBN: 978-988-14048-6-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

VI. CONCLUSION

This paper presents a new course module titled “Introduction to Secure Program Design” and reports our teaching results. This module covers computer security concepts; input validation flaws; buffer overflows; how to prevent input flaws and buffer overflows; and a laboratory exercise.

This module has been taught in the COMP 280 Data Structure class for undergraduate sophomore in the fall of 2018. The students’ survey results and feedback reflect that this module is very valuable. By teaching this module, students quickly obtained knowledge and understanding of the impacts of input validation flaws and buffer overflows and how to use secure program design to prevent input flaws and buffer overflow.

ACKNOWLEDGEMENT

This work is supported by National Science Foundation under the award number 1662469.

REFERENCES

[1] M. O. P. Frej, "Analysis of Buffer Overflow Attacks", Available:http://www.windowsecurity.com/articles/analysis_of_buffer_overflow_attacks.html.

[2] IEEE, “Avoiding the top 10 software security design flaws”, 2017, Available: http://cybersecurity.ieee.org/center-for-secure-design

[3] OWASP, "Category:Input Validation - OWASP", Available:https://www.owasp.org/index.php/Category:Input_Validation.

[4] M. Quinson, & O. Gérald, “A Teaching System to Learn Programming: the Programmer's Learning Machine”. *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, 2015.

[5] C. Theisen, L. Williams, K. Oliver and E. Murphy-Hill, “Software security education at scale”, IEEE/ACM International Conference on, 2016.

[6] K. Williams, X. Yuan, H. Yu and K. Bryant, “Teaching Secure Coding for Beginning Programmers”, Journal of Computing Sciences in Colleges (CCSC:MS), Volume 29, Issue 5, 2014.

[7] Wikipedia, "Buffer Overflow." the Free Encyclopedia. Web, September 2011, Available: http://en.wikipedia.org/wiki/Buffer_overflow.

[8] X. Yuan, L. Yang, B. Jones, H. Yu and B. Chu, “Secure Software Engineering Education: Knowledge Area, Curriculum and Resources”, *Information Security Education Journal*, 2015.

[9] H. Yu and N. Jones, “Secure Software Programming”, *Journal of American Business Review, Vol. 3, Num. 1,* December 2014.

ISBN: 978-988-14048-6-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)