INTRODUCING SECURE CODING CONCEPTS IN ENGINEERING PROGRAMMING

Kalyan Mondal
mondal@fdu.edu
Fairleigh Dickinson University
1000 River Road, T-MU1-01
Teaneck, NJ 07666

Abstract: Over the last several years, the world has witnessed numerous major cyber attacks resulting in stolen personal credit card numbers, leak of classified information vital for national defense, industrial espionage resulting in major financial losses, and many more malicious after effects. By using worms, viruses, Trojan horses, and various malicious tools, hackers have managed to make safe computing a more difficult task. This has resulted in introducing the concepts of cybersecurity in the education of computer science, information technology, and related fields. Cyber defense concepts are being introduced in traditional engineering and technology programs. To this end, we have developed a course module on "Secure Programming" to effectively teach the best practices of secure coding in Fairleigh Dickinson University's undergraduate engineering programming course. This paper presents the contents of this module and our experiences in teaching it in our ENGR 3200 "Advanced Engineering Programming" course during Fall 2012 semester for the first time. This course is taken by the engineering and engineering technology juniors learning C/C++ programming with engineering applications. This is the second course after the students completed procedural programming in MATLAB in their freshman year. The ENGR 3200 course starts out reviewing MATLAB coding, introducing the C procedural programming concepts and ends with C++ object-oriented programming. Class and lab exercises include coding both in C and C++. Our experience shows that teaching this "Secure Programming" module in ENGR 3200 help students not only to understand the impacts of insecure and not thoroughly tested code, but also to gain significant knowledge of safe programming practices. This module covers the computer security concepts, input/output data validation, secure string processing, and other best practices in secure programming. Students are not only taught how to mitigate such problems (practical aspect) but also the root cause (theoretical aspect) of their formation. We have developed a simple rubric to assess student learning of secure programming concepts in this course. Student learning outcome assessment shows that they get a good understanding of the risks and vulnerabilities and form basic ideas of secure programming to reduce such vulnerabilities. We are working towards possible introduction of similar concepts into lower level engineering and engineering technology programming courses.

Key Words: Programming, Secure Coding, Integer Overflow, Input Validation, Buffer Overflow

Introduction:

At our university, all engineering and engineering technology majors take a first course on programming – ENGR 1204 Programming Languages in Engineering (viz. http://www.fdu.edu/academic/1214bulletin/universitycoursedesc.pdf). This course involves procedural programming in MATLAB. Advanced programming course, ENGR 3200 is required to be taken additionally by the electrical engineering and electrical engineering technology majors. Programming in C and C++ are introduced in this course using the text book [1] and introduces both the procedural and object-oriented programming course topics. ENGR 3200 also is a prerequisite course for the EENG/EGTE 3288 Microprocessor System Design II which requires programming the Freescale HCS12 microcontroller using the C language. As such, this course is a prime candidate for introducing Secure Coding principles.

Secure coding concepts have been proposed and incorporated in some college curriculum [2 – 4] mostly in the computer science departments. However, due to the criticality of the cyber attacks and the resulting disruptions, all engineers must know the root cause of such issues and how to program defensively to minimize vulnerabilities in their code. Thus the concepts of secure programming need to be introduced in the normal engineering and engineering technology curriculum. Although there is a wealth of teaching material in the website [3], the right place of injecting such modules is left to the choice of the instructors. In this paper, we outline which Secure Coding concepts were introduced and at what point in the ENGR 3200 course. The main idea is not to disrupt the flow of topics in this course, but to introduce the security concepts and examples integrated throughout the course. Final discussions culminated in a separate module that ties all the concepts to the actual threats and vulnerabilities.

This paper is divided into the following sections: Integer Overflow, Input Data Validation, Buffer Overflow, File I/O, String Processing, and Secure Programming Module. Finally, we summarize the student outcome of answering secure coding related Practice questions in the final examination in the Conclusion section. All examples in this paper were verified using Microsoft Visual Studio 2010 Premium.

Integer Overflow:

The problem of integer overflow occurs when the user-entered data exceeds the range of values acceptable for the relevant variable. Due to overflow of buffer saving the integer data, it can lead to unpredictable behavior of the program and potential security risk. This topic was discussed, in our course, right after the concepts of control structure ("if – else") was introduced (viz., Section 4.2 of [1]). By this time, students already learned about arithmetic operations, variable declaration statements, assignment operations, and interactive input. So they could write simple programs using "int" and "double" variable types and control structures. At this point, they were shown the following example of integer overflow and a solution to the problem was also discussed.

The C++ program in Fig. 1 is adapted from [2] and asks for user input of an integer. If the entered value is even, the number is divided by two, else it is doubled and one is added to it. The relevant variable "num" is declared to be of type "short int" whereby its value is limited between -32,768 and 32,767. Since there is no input data validation, the program may result in an integer overflow whereby for a large positive integer input (e.g., 16385), the result may come out to be negative (-32765). The correct result should have been 32,771, but due to arithmetic overflow and two's complement wrap-around, the result became negative. This particular example not only exemplifies the issue of Integer Overflow but also points out the vulnerability introduced due to a lack of proper input data validation.

The problem can be resolved in two different ways. First, it can be fixed by adding an input data range check selection structure as shown in Fig. 2. This allows the program to abort with a message, if the user data entry exceeds the programmed limit. The second approach would be to change the variable declaration of "short int num" by the "long num". This approach increases the allowable number range to a very high value, decreasing the probability of Integer Overflow. In this course, we emphasized the first method shown in Fig. 2.

An excellent set of coding guidelines to avoid Integer Overflow in a C/C++ program is included in [3]: http://cis1.towson.edu/~cssecinj/modules/cs2/cs2-integer-error-c.

Input Data Validation:

Several input data related vulnerabilities are not uncovered unless the program is tested thoroughly with multiple test scenarios. This particular point is emphasized in our course right from the beginning of writing C/C++ programs involving interactive input. The C++ program in Fig. 3, adapted from [3], allows the users to find course letter grades after entering a test score value. It is a simple program using nested if-else loop and was discussed while covering Section 4.3 of [1]. It works properly for all integer test score values. However, due to a lack of input data range check, the program will allow the user to enter invalid negative test scores or test scores over 100.

A fix to the program can be done by incorporating an "if-else" selection structure similar to what was done in Fig. 2. This will abort the program when erroneous data is entered by the user. Instead, we discussed a fix involving a "while" loop structure that allows the user to re-enter the correct data without aborting the program. Such a fix could be introduced after students had mastered loop structures in Section 5.2 of the text [1]. The program with input data validation (data range check) and interactive input is included in Fig. 4.

Additional input data validation technique outlined in Section 9.2 of the text [1] also was covered in detail. In this treatment, the user input data is always captured as a string literal. This string is analyzed to ascertain its nature (such as integer) prior to converting it to an integer and using it in the rest of the program. Invalid user input is corrected using a looping mechanism. Still the data range check is necessary to ascertain that the entered data is valid.

Buffer Overflow:

A buffer overflow is possible when the length of some input data (usually some input string or integer) is not correctly checked, before it is copied to some buffer on the stack. If the target buffer is smaller than the input data, the bytes on the stack above the target buffer get overwritten. Students were told about this problem after they had mastered the concepts of arrays as described in Chapter 7 of the text [1].

An easy mechanism of causing inadvertent buffer overflow is shown in Fig. 5. This program adapted from [4], shows an inadvertent mistake in using the "for" loop structure to populate the "buffer[]" array. The "for" loop runs through 100 times as specified by incrementing the loop counter "i". The next assignment statement tries to assign the value of NULL to the array element "buffer[100]" which was does not have any compiler pre-allocated space. So the program does compile and build, but rightly aborts with runtime exception as shown in Fig. 5.

A second example is shown in Fig. 6.This example is adapted from [3]. In here the "for" loop structure populates the "buffer[]" array beyond its allocated space of 10 integer values resulting in buffer overflow. The code compiles and fails with runtime exception. The data saved in the buffer seem to be not corrupted, but this result is compiler dependent. In any case, the program aborts and is vulnerable from the point of view of "buffer overflow" attacks. The fix to the code is simple, keep the array dimension and loop count maximum to the same value. This can be readily done by following simple modifications shown in Fig. 7.

File I/O:

Opening a file for data read or write operation is quite common in programming. Chapter 8 in [1] covers the topics of I/O Streams and Data Files. Students learned how to use input and output stream functions and open files for input output operations. For secure coding, after a file has been opened, it should only be accessed via its file handle or file descriptor whenever possible. Time and again, we emphasized students to close an opened file as soon as it is no longer needed for data I/O. We mentioned certain types of attacks known as "Time−of− check, time−of−use" (TOCTOU) can be avoided by being meticulous about file open and close operations.

String Processing:

Several string and character manipulation functions are offered through standard libraries and class libraries in C and C++ programming. ANSI C <string.h> library [5] includes a set of functions such as "char *strcat", "int strcmp", "char *strcpy", etc. which are known have vulnerabilities. These functions could lead to buffer overflows and can potentially lead to devastating Denial of Service (DoS) attacks. Students were told never to use these functions, instead, they should use similar overall functionality functions: "char *strncat", "int strncmp", "char *strncpy", etc. These functions limit operations between strings of a specified length and avoid "buffer overflow attacks."

We also caution students using the C++ "string" class processing functions like "void insert", "void replace", etc. since they allow direct memory access and can potentially lead to memory corruption. Programmer's motto always should be "defensive programming" by not going "out of bounds" in memory usage. This topic was covered after "string" class and the relevant functions were discussed per Section 9.3 of [1].

Secure Programming Module:

This module was introduced after students had gone through the problems of Integer Overflow, Input Data Validation, File I/O, and String Processing oriented vulnerabilities. It included simple descriptions of Denial of Service (DoS) attacks, Buffer Overflow attacks, and TOCTOU attacks. It also introduced a list of safe programming practices to be used by C and C++ programmers. A set of C++ specific security measures provided to the students is shown below.

- Declare variables and methods to be private whenever possible. (Principle of Least Privilege)
- Make public access to variables only possible via accessor methods (get/set methods).
- Immutable objects are objects whose state (value of member variables) cannot be changed after construction. It is a good (safe) coding rule to reduce the number of modifiable variables as much as possible.

We pointed out additional specific measures taught earlier in this course that can avoid some of the cyber attacks.

Conclusion:

In this paper we have summarized the injection of secure programming concepts into an advanced programming course taken by electrical engineering and engineering technology majors. The student learning outcome was assessed partially in this first offering through a set of practice questions in the final examination. Out of 12 students taking the test, only one student obtained a failing score in this segment. This student happened to fail the course as well. The average score of the class for this segment of learning outcome was nearly 82%, including the grade of the failing student.

In the next offering of the course, a more comprehensive rubric will be developed and used for student learning outcome assessment of Secure Programming principles. In future, we plan to develop and inject some of the security concepts in the first programming course taken by all engineering and engineering technology majors in our university.

References

[1] Gary J. Bronson, *C++ for Engineers and Scientists*. Boston, MA: Cengage Learning, 2013.

[2] Huiming Yu, Nadia Jones, Gina Bullock, and Xiaohong Yuan Yuan, "Teaching Secure Software Engineering: Writing Secure Code," *2011 7th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR)*, pp. 1 – 5, Nov. 2011.

[3] Security Injections @Towson University URL: http://cis1.towson.edu/~cssecinj/modules/cs0/input-validation-cs0-c accessed on 3/13/2013.

[4] Chen Bo, Xu Da-wei, Gao Si-dan, and Yu Ling, "Cultivating the Ability of Security Coding for

Undergraduates in Programming Teaching," *Proceedings of 2009 4th International Conference on Computer Science & Education*, Nanning, China, pp. 1425 – 1430, July 2009.

[5] Delores M. Etter, *Engineering Problem Solving with C*, 3rd Edition. Upper Saddle River, NJ: Person Prentice-Hall, 2005.

Figures with Captions

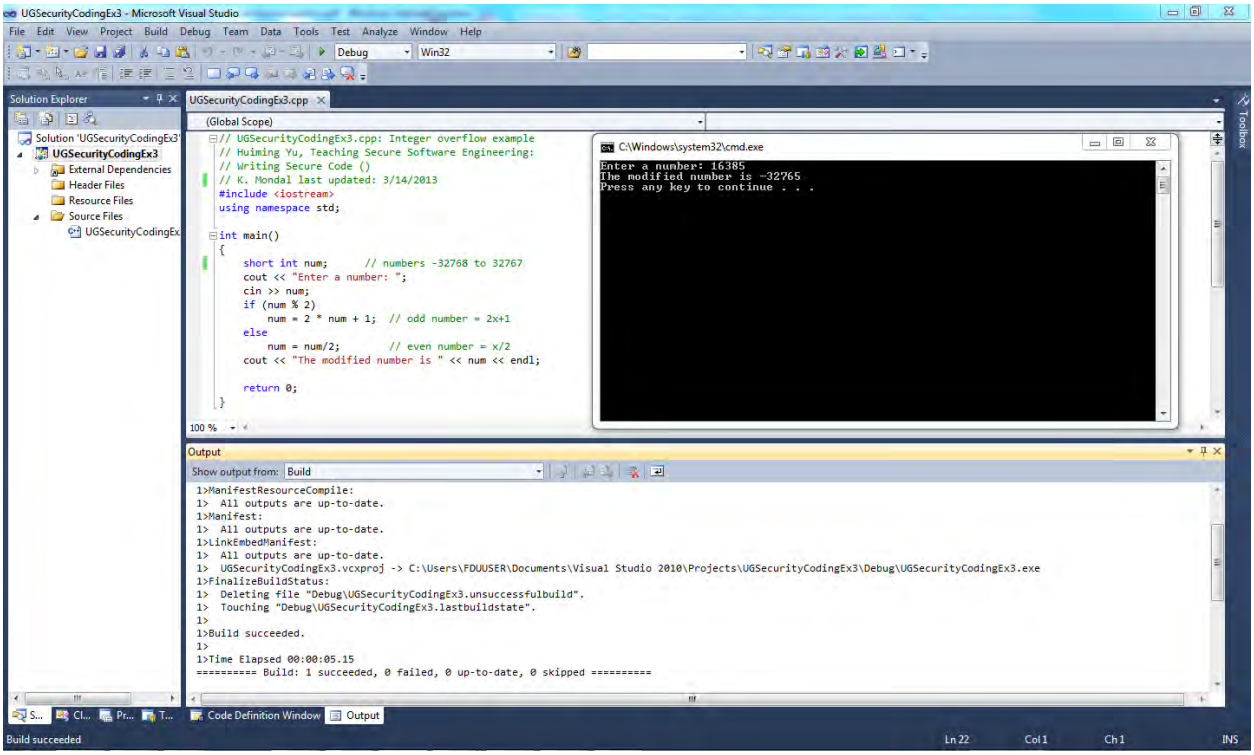

Figure 1: An example program with potential Integer Overflow.

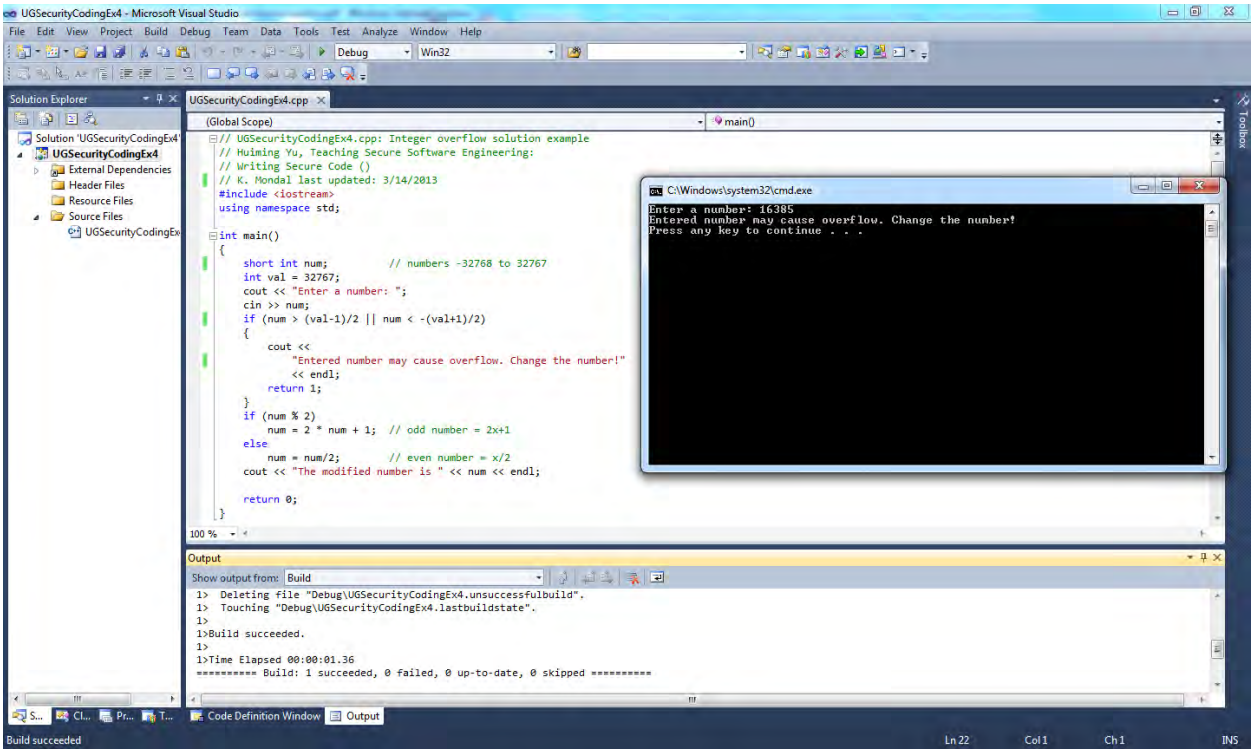

Figure 2: Modified example of Fig. 1 eliminating Integer Overflow.

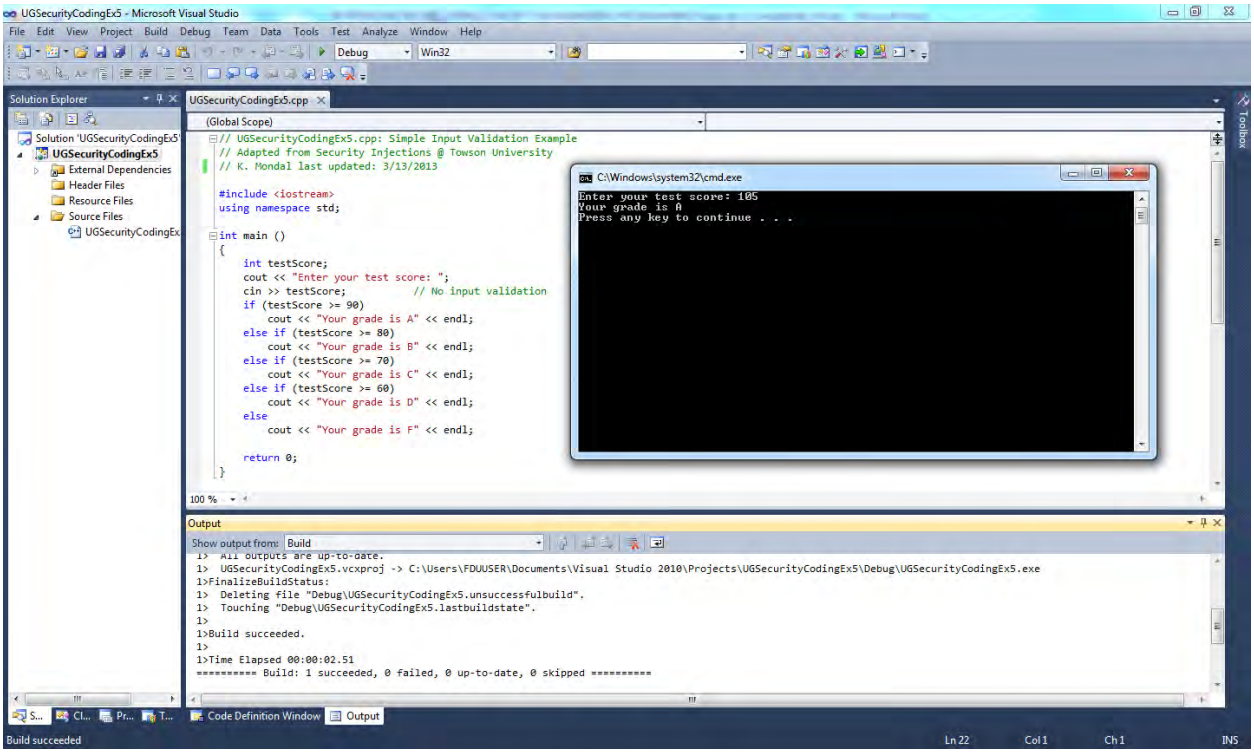

Figure 3: An example program without input data validation.

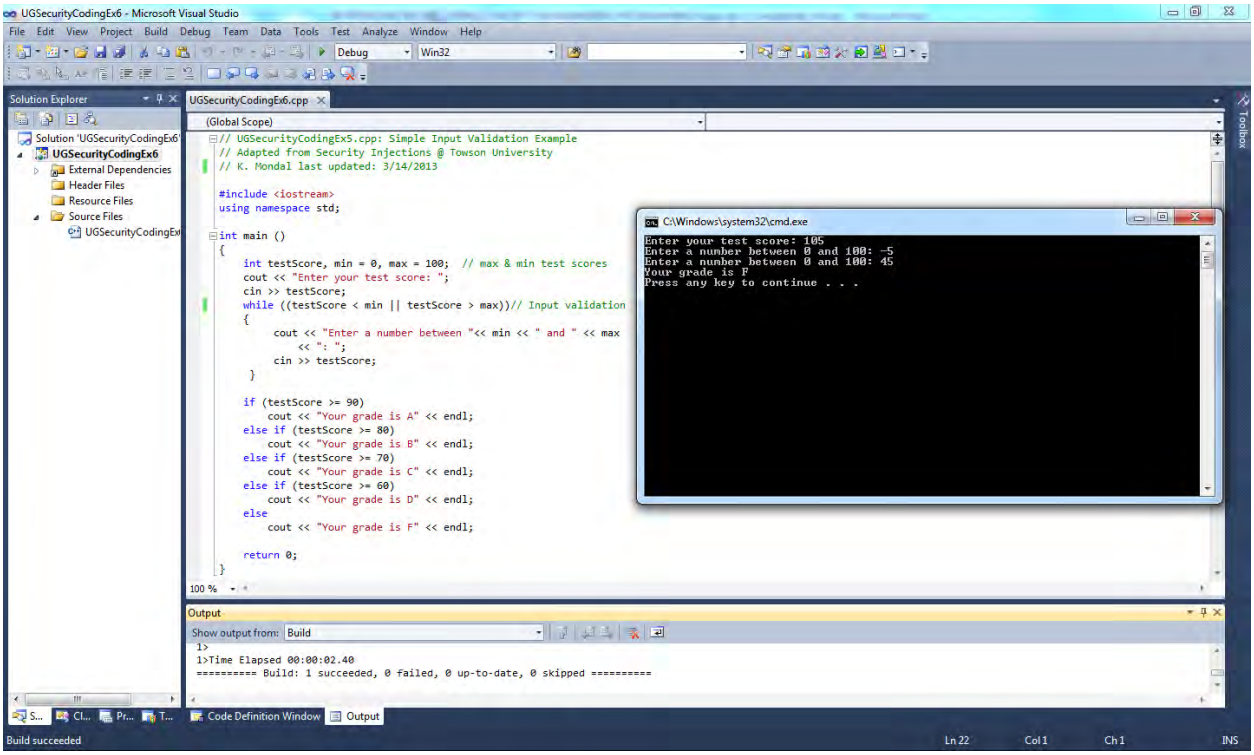

Figure 4: Program in Fig. 3 enhanced using data range check and interactive input

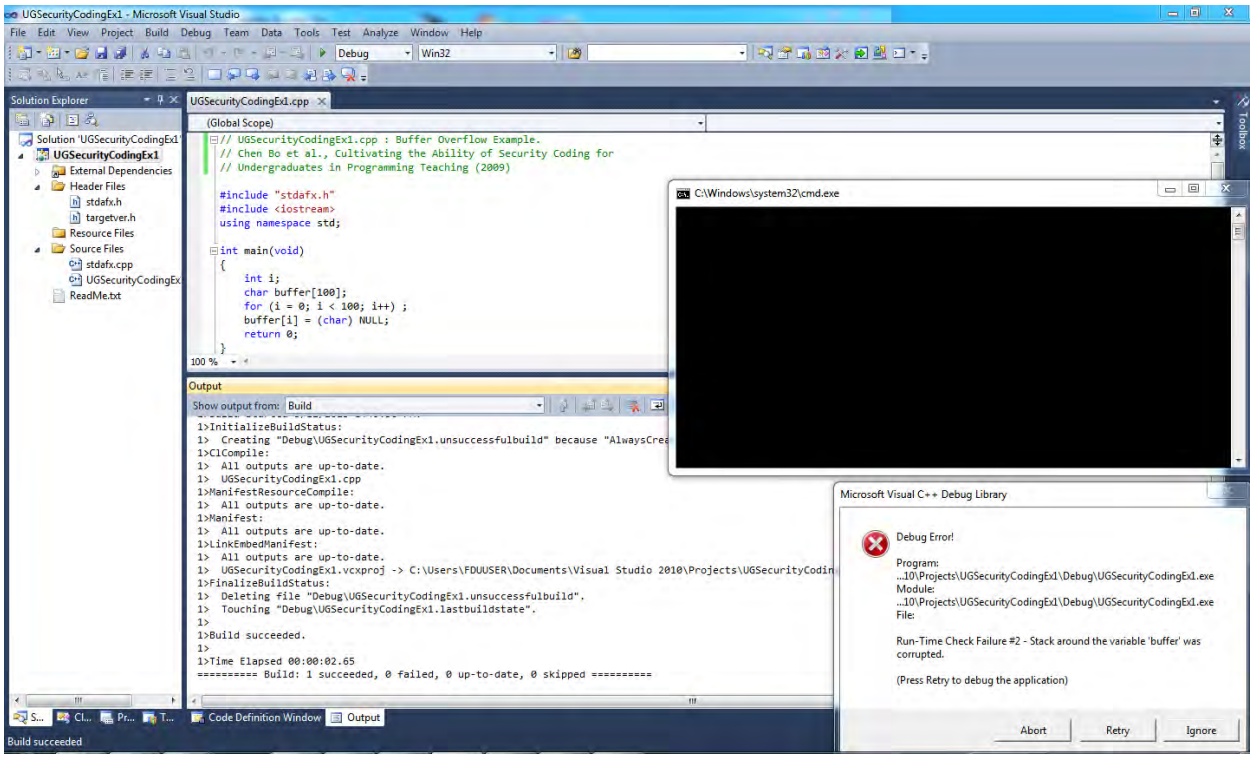

Figure 5: An example of buffer overflow caused by inadvertent programming error.

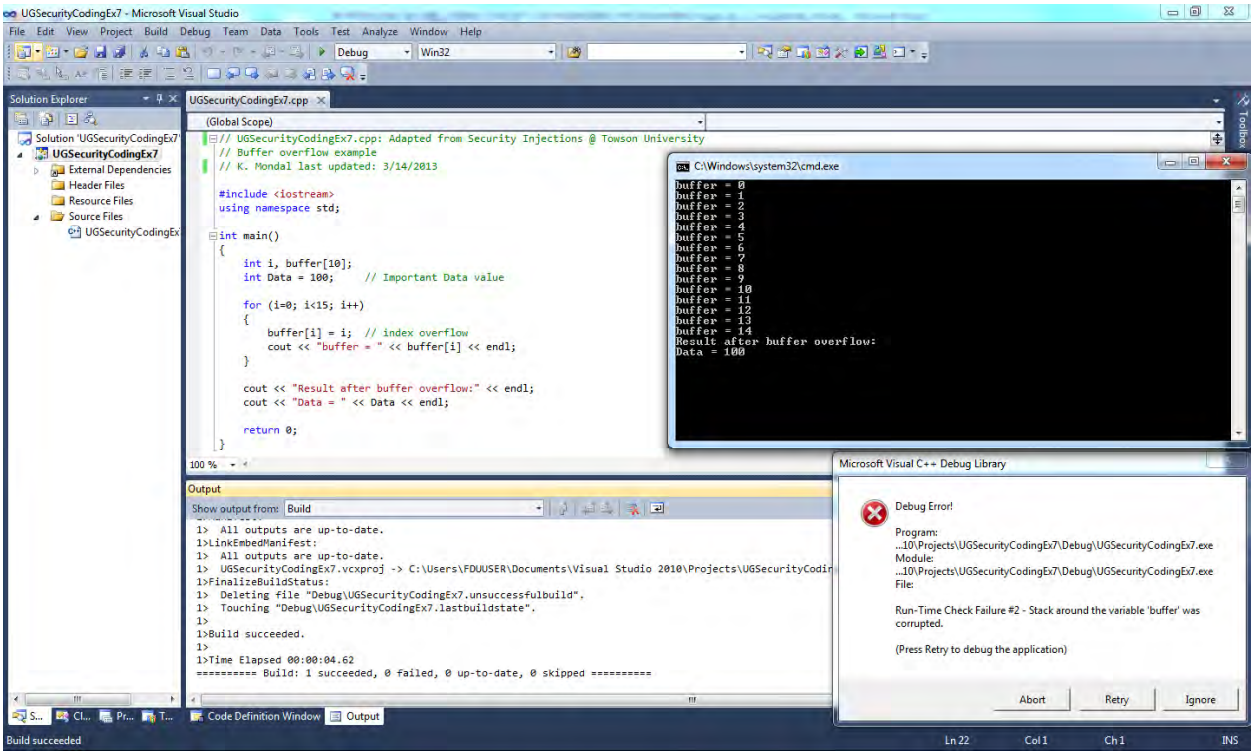

Figure 6: Another example of buffer overflow due to improper array size declaration.

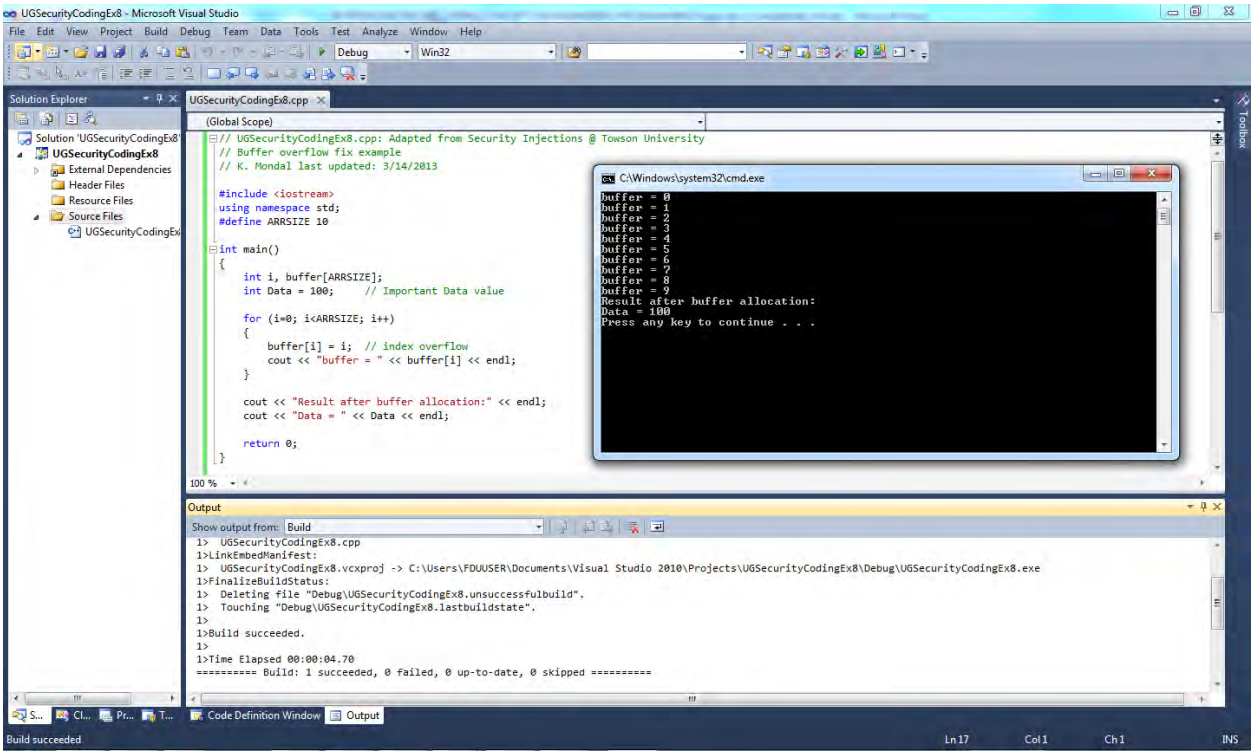

Figure 7: Fixed buffer overflow example from Fig. 6.