What Dick and Jane Don't Know About Integers

Laurie Werner
wernerla@muohio.edu

Charles Frank
frank@nku.edu

ABSTRACT

The integer data type is ostensibly very simple, but integers can easily overflow in a simple program. A malicious user can manipulate an unchecked integer input to overflow which can produce a security breach. An integer overflow can cause a program to crash. In recent years, integer overflows resulted in more than two hundred recorded vulnerabilities. Integer overflow is a challenging topic to address when teaching C/C++ or Java in an introductory software development course. Most novice students are unaware that simple integer input or calculations can generate errors, or worse yet, silently deliver vulnerability in a system. This paper describes laboratory exercises that inform students about the nuances of integer behavior and how these can lead to security vulnerabilities. We illustrate techniques that educators can use to teach students to discover integer overflows and replace them with robust code. Even at the introductory level, we can reinforce a secure coding frame of mind such that our students will never blindly trust user input or perform calculations that generate integer overflows.

Keywords: integer overflow, security education, introductory programming, secure programming, Java, C, C++

1. INTRODUCTION

Howard, LeBlanc, and Viega (2005) devote a chapter to integer overflows in their book on the most common security programming flaws. They cite several security problems such as a flaw in the Windows script engine that could allow an attacker to use an integer overflow to execute arbitrary code via a malicious web page or HTML e-mail. Howard, et al (2005), quotes Theo de Raadt, of OpenBSD fame, who claims that integer overflows are "the next big threat." A search of The National Vulnerability Database (2006) indicates that more than 200 vulnerabilities have been caused by integer overflows since January 2003.

Liang (2005) and Savitch (2006) are typical introductory Java programming textbooks. Early in both texts, integer data types are introduced, their ranges of valid values are given, and arithmetic operations are described. Neither textbook presents the fact that an arithmetic operation on integers may overflow its maximum value. Savitch (2006) sprinkles the text with many useful "pitfalls," which warn students about how to avoid errors, but none of these alludes to integer overflows.

Yet an integer overflow in Java occurs silently, without throwing an exception. Since the ISO C99 standard defines the appropriate behavior for the integer overflow case to be "undefined behavior," (Walden, 2005) presumably because most programming languages provide no mechanism to test the overflow bit of the flags register, our students will need to be aware of integer overflows even before they are aware of the security vulnerabilities that one may produce.

In his review of current security education, Pothamsetty (2005) observes that writing secure code "does not require a lot of security training and knowledge" but requires following correct software development techniques. Coupled with the well-documented success of various types of laboratory activities in teaching programming fundamentals (Beck 11; McKinney, 2006), the next practical step to incorporate security awareness in the software development portion of the IS curriculum is via early and frequent laboratory experiences.

Programs that exhibit unexpected behavior engage student interest, and provide instructors with a valuable opportunity to introduce the concepts of robustness and trustworthy computing. When integers overflow, results that should be positive become negative. It is beneficial for students to learn early not to trust user input and to practice defensive programming early. Laboratory activities are the perfect setting to incorporate defensive programming, while teaching basic concepts, such as writing simple methods, learning loop syntax and logic, or sampling the Math class methods.

2. SECURITY VULNERABILITY: INTEGER OVERFLOW

Howard, et al (2005), presents each of the 19 Deadly Sins of Software Security in a two-part format. For each "sin", they describe with examples, they provide "redemption steps." One of these nineteen deadly sins is integer overflow. Because a finite number of bits represents integer data types, integers can overflow in all common languages. The consequences of integer overflows are more serious in C/C++ since they can lead to buffer overflows and arbitrary code execution. A common problem is mixing signed and unsigned integers. The conversion rules for integers of different sizes and for signed and unsigned integers are described in Howard (2005) and Seacord (2006). Although the rules are too detailed for an introductory course, examples and explanations of overflow behavior are appropriate for novice programmers.

Consider the following C function (Wilson, 2006) that takes a filename as input, removes the extension by removing the last four characters, and displays results. For example, the file "a.doc" would be displayed as "a".

```
void StripExtension (char * filename)
{
unsigned short int newSize = strlen(filename)  - 4;
        char * buffer = (char *)malloc(newSize + 1);
        strncpy(buffer, filename, newSize);
        buffer[newSize] = '\0';
        printf("%s", buffer);
        free(buffer);
}
```

What would happen if StripExtension were called as follows, without the file extension?

```
StripExtension("a");
```

In this case, newSize = 0xffffd = (1 minus 4), which is the bit representation of -3. Since newSize is an unsigned short integer, this value is 65533. The function creates a 65534-byte buffer.

Novice students generally assume that users enter data in the correct format. Without instructor guidance, they are unaware that when the user's input string does not end with a period followed by a three-character extension, StripExtension assigns newSize an overflow value. At an introductory level, the redemption step to prevent the overflow could be to check for the extension, and have the function display an error message or return an error value if the extension did not exist.

3. AWARENESS OF THE SIN: STUDENTS GENERATE INTEGER OVERFLOWS

Here we introduce the budding potential of defensive programming in laboratory activities, while also teaching fundamental programming concepts. Java is now more common in beginning and intermediate programming courses than C/C++. Although Java is generally less vulnerable to security problems, unsigned integers are still subject to overflow. In a lab activity to practice writing and calling Java 5 static helper methods, it is possible to alert students to the existence of integer overflows.

A few pre-lab questions set the stage:

1. What is 2^{32}? 4,294,967,296
2. What is 2^{31}? 2,147,483,648
3. What is 32768 * 65536? 2,147,483,648
4. What is the largest int in Java? 2,147,483,647

5. What is the smallest int in Java? -2,147,483,648

First, the students write a simple method to display the powers of two from 2^0 to 2^{32} using integers and doubles. The method has no arguments and returns no values, but will force the students to recall how Math.pow() works and to use a loop in a method. Here is an actual student pair's solution, with line numbers:

```
1. public static void displayPowers(){
2. double powerOfTwoAsDouble = 0.0;
3. int powerOfTwoAsInt = 0;
4. for (int power = 0; power <=32; power++){
5. powerOfTwoAsDouble =  Math.pow(2,power);
6. powerOfTwoAsInt = (int)powerOfTwoAsDouble;
7. System.out.printf("%3d   %5d ",power,
                         powerOfTwoAsInt);
8. System.out.printf("%13.0f\n",powerOfTwoAsDouble
                         );
9.  }
10.   }
11.   public static void main(String[] args) {
12.         displayPowers();
13.  }
```

As the students observe the output, they are astonished. The last few lines of output are:

```
 30  1073741824    1073741824
 31  2147483647    2147483648
 32  2147483647    4294967296
```

To some, it appears that the loop "sticks," as one student commented. Some students suggest rounding instead of casting the double in line 7. They soon realize that Math.round() returns a long, but the lab specifies using int data type. Thus, changing line 6 to cast the result produces slightly different, yet still startling output for the last few lines. Changing line 6 to: powerOfTwoAsInt = (int)Math.round(powerOfTwoAsDouble); changes the last three lines of output to:

```
30  1073741824    1073741824
31  -2147483648   2147483648
32  0   4294967296
```

Second, students write a static method to multiply two integers and return the result and test it by calling it in a main method:

```
public static int mult(int a, int b ){
      return a * b;
}
```

```
public static void main(String[] args) {
      Scanner kb = new Scanner(System.in);
      System.out.println("Enter two integers separated by
        a space: ");
      int firstInt = kb.nextInt();
      int secondInt = kb.nextInt();
      int prod = mult(firstInt,secondInt);
      System.out.println("The product is "+prod);
      }
```

When the students enter the operand values from preliminary question #3, the result is:

```
Enter two integers separated by a space:
65536 32768

The product is -2147483648
```

This is not what the students expect. They are puzzled. They calculated the correct answers by hand, so they know the output is wrong. Before discussing why, they run the program with a few more pairs of larger integers:

```
Enter two integers separated by a space:
65535 65536
The product is -65536
```

```
Enter two integers separated by a space:
65535 65537
The product is -1
```

Students want to know "Why do both programs produce such bizarre numbers? What is happening here?" At the very least, students are impressed with the untrustworthiness of the output. From an instructor's viewpoint, this is value added to the normal lab time intended to practice writing methods with loops, formatting output, and casting and to observe Math method behaviors. Before the students leave the lab, they attempt to answer three questions from a Java perspective rather than an algebraic one.

1.What two non-zero integers x and y satisfy the equation x*y=0?

2.What negative integer (-x) has no corresponding positive integer (x)?

3.List two positive integers x and y, such that x + y < 0.

The following class meeting begins with the three questions. Depending on the level of the students' Java skills, the instructor can take the discussion from the unusual lab activity and three peculiar questions to another level of defensive programming, such as input validation: how to develop methods for checking user input. Another possibility is to use the activity as a springboard for a dialogue about the details of integer storage, or the complex issues of integer arithmetic. (Seacord, 2006; LeBlanc, 2004; SafeInt, 2004)

4. REDEMPTION STEPS: TECHNIQUES TO PREVENT INTEGER OVERFLOWS

Consistent with the "deadly sins" framework of "redemption steps" and the knowledge that input validation is one of the top two vulnerabilities in a 2005 CERIAS report (Pothamsetty, 2005; Gopalakrishna, 2005), one of the first redemption steps is to validate input before performing computations. However, input validation is only part of the Integer overflow story. Computations can generate integer overflows within unsuspecting programs. In this section, we present a novice student group's attempt to validate integer input, and describe three techniques that prevent integer overflows. The first prevention technique describes how to check the results of integer arithmetic operations to insure that overflow does not occur. The second demonstrates Java's BigInteger (Java, 2006) class, which can represent arbitrarily large integers. Finally, we illustrate David LeBlanc's SafeInt class (SafeInt, 2004) for C++.

Check User Input

Students build some algorithm skills, and practice using the String class methods by developing a method that converts String input to integer input. In Java, we use a wrapper class method to convert string input to an integer:

```
int value = Integer.parseInt(input);
```

However, this method call may cause an exception if the variable input is not the String representation of an integer. Even if the input is all digits, it may not convert to a valid integer. It is a worthwhile assignment or lab exercise for novices to validate the String input before sending it to Integer.parseInt(). The following code block is a lab group's first attempt at validating the input. The students were in the eighth week of their first programming course in Java.

```
public static String validateIntegerInput(){
   System.out.print("Enter an integer: ");
   Scanner kb = new Scanner(System.in);
   String newIntString = kb.nextLine();
   boolean isNegative = false;//assume positive
   boolean lengthOK = true; // count the characters
   boolean allDigits = true; //
   // check for length
   if (newIntString.length()>11){
      System.out.println("Number is too many digits");
      lengthOK = false;
   }
   else if (newIntString.charAt(0) == '-'){
      isNegative = true;
   }
   else if (newIntString.length()==11
                &&isNegative == false){
      lengthOK = false;
   }
   if (lengthOK){
      int i = 0;
      if (isNegative)
         i = 1;
      for ( ; i<newIntString.length();i++){
         char temp = newIntString.charAt(i);
         if( Character.isDigit(temp) == false ){
            allDigits = false;
         }
      }// end for
   }
   if (allDigits && lengthOK)
      return newIntString;
   else
      return "error";
}
```

Although this attempt is not entirely robust, the lab group integrated the expertise they had at the time. Once we introduce defensive programming, we can encourage an attitude of competition among the lab groups throughout the semester to generate the most robust program possible for the students' expertise at the time. We are hopeful that the defensive programming posture endures as the students' skill set increases in later courses. At this writing, students' program solutions were more robust than expected for eight weeks into an introductory course. With some encouragement in the form of a better grade, most students began to import and use the above static method that was collaboratively developed in class in their program assignments.

Check Integer Arithmetic

Validating input as integer provides the first step, but what about simple calculations such as x + y? How do we prevent a result from overflow? Students typically assume that x and y are positive, and thus test x+y> MAX_INT, without realizing that they have already caused the overflow if x added to y is larger than the maximum integer. In Java, beginning students typically produce a code snippet something like this when asked to error check the addition of two very large integers.

```
int x = Integer.MAX_VALUE;
int y = 1;
if ( (x+y) > Integer.MAX_VALUE){
      System.out.println("Sum is too large. ");}
else{
      int sum = x+y;
      System.out.println("Sum is "+sum);}
```

Students assume that the output will be "Sum is too large." But the above code segment surprises the novice programmer with:

```
Sum is -2147483648
```

If we introduce the concept that for two unsigned valid integers, we can check the sum by checking x > Integer.MAX_VALUE – y, then we have a better result. Modifying the above if statement to

```
if ( x > (Integer.MAX_VALUE - y))
      System.out.println("Sum is too large ");
else{
      int sum = x+y;
      System.out.println("Sum is "+sum);
}
```

produces this output:

```
Sum is too large
```

For multiplication of Java integers, this example demonstrates an algorithm that provides some protection from integer overflow:

```
int x = Integer.MAX_VALUE/2;
int y = 3;
   if ( x > (Integer.MAX_VALUE/y))
      System.out.println("Product exceeds maximum
                     allowed integer. ");
   else{
      int prod = x*y;
      System.out.println("product is "+prod);
   }
```

These two algorithms work nicely for unsigned integers, which we foster whenever possible. Alas, Java provides no unsigned integers. Given that these two defensive steps are not the entire solution to integer overflow when adding or multiplying, they provide a petite step in the trek to the defensive programming summit.

Java's Big Integer Class

Another example of integer overflow that novice programmers can easily understand is the following Java program that prints factorials. Factorials grow rapidly in size. Although this program uses a 64-bit long integer, 21! overflows and the output values become negative. Here is a simple iterative program to compute factorial:

```
public class Factorial{
   public static void main(String args[])
   {
      long product = 1;
      for(int i = 1; i <= 21; i++){
            System.out.print(i);
            System.out.print("! = ");
            product *= i;
            System.out.println(product);
      }// end for
   }// end main
}// end factorial
```

As expected, the program's output overflows at 21!:

```
1! = 1
2! = 2
3! = 6

....
20! = 2432902008176640000
21! = -4249290049419214848
```

Java provides a BigInteger class (Java, 2006), which has a capacity for as large an integer as necessary to accommodate the results of an operation. The following Java program is a revision of the factorial program using the BigInteger class. It correctly displays 21!. A solution using BigInteger follows:

```
import java.math.BigInteger;
public class BigFactorials{
   public static void main(String args[]) {
      BigInteger product = BigInteger.ONE;
      BigInteger index = BigInteger.ONE;
      for(int i = 1; i <= 21; i++){
            System.out.print(i);
            System.out.print("! = ");
            product = product.multiply(index);
            System.out.println(product);
            index = index.add(BigInteger.ONE);
      }//end for
   }//end main
}//end class
```

The program's output correctly displays 21!:

```
1! = 1
2! = 2
3! = 6
…
20! = 2432902008176640000
21! = 51090942171709440000
```

In situations like this, where very large integer values are possible, Java programmers can avoid integer overflows by converting to the BigInteger class. Once students are familiar with using more of the basic Java classes, it is reasonable to introduce and encourage use of the BigInteger class as an application may warrant.

SafeInt class for C++

Since handling an integer safely is quite complex, David LeBlanc (2004) provides redemption for C++ integer overflows in his SafeInt class for C++. He describes the issues involved in safe integer arithmetic in depth, and concurs with Theo deRaadt about the importance of eliminating integer overflows. LeBlanc comments "As we reduce our dependency on unsafe string handling calls, the arithmetic used to determine buffer lengths becomes the weak link, and thus the area attackers will attempt to exploit. It is also possible for integer overflows to result in logic errors not related to string handling—the effects of the logic errors have historically ranged from crashes to escalation of privilege." (2004)

Seacord (2006) presents this illustration of the SafeInt class. When an integer operation overflows, the SafeInt code throws an exception. Here is an example of addition using the SafeInt class:

```
SafeInt<T> operator +(SafeInt<T> rhs) {
return SafeInt<T>(addition(m_int,rhs.Value()));
}
int main(int argc, char *const *argv) {
  try {
    SafeInt<unsigned long> s1(strlen(argv[1]));
    SafeInt<unsigned long> s2(strlen(argv[2]));
    char *buff = (char *) malloc(s1 + s2 + 1);
    strcpy(buff, argv[1]);
    strcat(buff, argv[2]);
  }
  catch(SafeIntException err) {
    abort();
  }
}//end main
}
```

5. CONCLUSION

Introducing integer overflows early in programming courses persuades students to think carefully about the validity of user input, and to question results of computations. It demonstrates and reinforces several concepts that are generally desirable in an introductory course, while simultaneously contributing to a methodology of defensive software development that is essential to decreasing the vulnerabilities in the software of the future. In his analysis of whether the content of current specialized security courses can mitigate security vulnerabilities, Pothamsetty (2005) concludes that current security courses "provide plaster around the sore but will not contribute towards preventing the disease." Since security specialists do not write most commercial software, the true prevention is in core undergraduate courses that teach secure software practices. Let us start with the awareness and control of integer overflows and move forward.

6. REFERENCES

Beck, L., Chizhik, A., McElroy, A. (2005) "Cooperative Learning Techniques in CS1: Design and Experimental Evaluation", Proceedings of the 36th SIGCSE Technical Symposium On Computer Science Education, St. Louis, Missouri, USA, March 1-5, Pp 470-474.

Gopalakrishna, Rajeev and Eugene H. Spafford (2005) "A Trend Analysis of Vulnerabilities" Technical report, CERIAS, Purdue University. CERIAS TR 2005-05.

Howard, M., LeBlanc, D., and Viega, J., (2005) 19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them. McGraw-Hill/Osborne, chapter 3.

Java 2 Platform Documentation (2006) http://java.sun.com/j2se/1.5.0/docs/api/.

LeBlanc, D, (2004) "Integer Handling with the C++ SafeInt Class" http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure01142004.asp

Liang, Y. (2005) Introduction to Java Programming: Comprehensive Version, 5th edition, Pearson/Prentice Hall.

McKinney, D., Denton, L. (2006) "Developing Collaborative Skills Early in the CS Curriculum in a Laboratory Environment", Proceedings of the 37th SIGCSE Technical Symposium On Computer Science Education, Houston, Texas, USA, March 1-5, Pp 138-142.

National Vulnerability Database, (2006) http://nvd.nist.gov/nvd.cfm

Pothamsetty, Victor (2005) "Where Security Education is Lacking", Proceedings of the 2nd Annual Conference on Information Security Curriculum Development, Kennesaw, GA, 2005, Pp 54-58.

SafeInt Class Code (2003) http://msdn.microsoft.com/library/en-us/dncode/html/secure01142004_sample.txt

Savitch, Walter (2006) Absolute Java, 2nd Edition, Addison Wesley.

Seacord, R. (2006) Secure Coding in C and C++, Addison Wesley, chapter 5.

Tsipenyuk, K., Chess, B., McGraw, G. (2005) "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors", IEEE Security and Privacy, Vol. 3, No. 6, Pp 81-84

Walden, James., Charles Frank, and Laurie Werner (2005) "Secure Programming Workshop", Journal of Computing Sciences in Colleges, Vol. 21, No. 1, Pp 134-135

Wilson, B. (2006) "Under the Hood: How an Attack Works", Dayton RISC 2006, March 15, Dayton, Ohio