Chapter 10
Numeric Errors: Integers

Revision 3.5, December 2025.

Objectives

- Understand the potential pitfalls associated with integers.
- Understand the causes of those problems.
- Learn how to mitigate integer problems.

10.1 Motivation

Improper integer computation is a common cause of security issues that manifest in many ways. Unfortunately, due to limitations of hardware and historical reasons, many programming languages silently allow many of these errors to go undetected.

Programmers naturally assume that integer conversions and calculations will be accurate and obey the laws of mathematics, but of course since computers express integers as binary values contained in a limited number of bits, the opportunity for errors to occur is present. Even experienced programmers who know better may overlook bugs due to wrong assumptions, faulty reasoning, or simply forget to use necessary caution.

Once miscalculation creeps into code, it can manifest as a security vulnerability in many ways, even far downstream from where the original misstep happened. While the concept of numeric errors in code is simple, these operations are so common and pervasive that avoiding all the traps in practice is an ongoing challenge.

10.2 Integer Arithmetic Fundamentals

To understand how integer conversion and calculation errors happen, recall that many languages are based on integer types that are represented as fixed-length bit strings. The C language is the common ancestor of many languages, and it includes a number of sizes of integers including the `char` type, and what is important to stress for our purposes here is that the language specification includes many subtle distinctions and explicitly leaves unspecified (for compiler designers to determine) the details of using these types correctly.

Consider the example of the number 256 (2^8) represented as a 16 bit integer, with the 1 followed by eight zeros on its right.

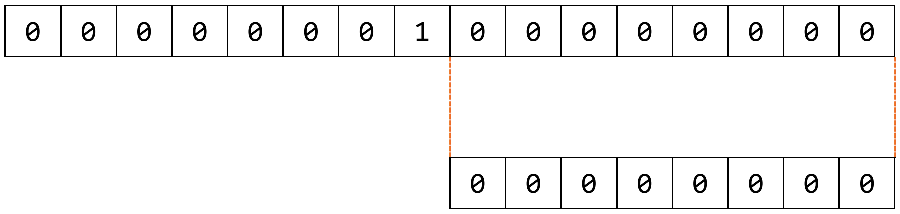


Figure 1: Truncation of 16 bit Integer to 8 Bits, Caused by Truncation

When this nonzero value is converted (cast) to an 8 bit integer it will be truncated, losing the leftmost 8 bits and become zero.

Computation within the same type can also lead to overflow and surprising results. If we had multiplied the value 256 by itself, the result would be too big to represent as a 16 bit integer and been truncated to zero.

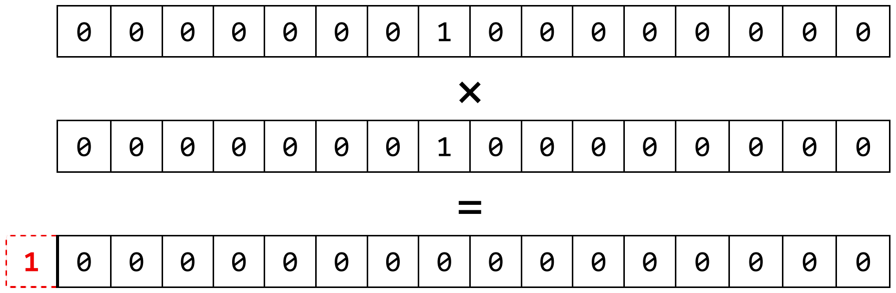


Figure 2: Overflow Caused by Multiplying Two 16 Bit Numbers

Since signed integers are represented as two's complement[1], even more surprising results are possible. The details of two's complement are outside our scope, but here we present a few characteristics:

- The sign bit is the high order (leftmost) bit.
- There is only one representation of zero, all bits zero. This representation is different from one's complement[2], where there is (annoyingly) both a positive and negative zero value.
- To form the additive inverse of a number, you take the bit-wise complement and then add one. So, for a 4 bit two's complement number, the additive inverse of `0001` is `1111`.
- The range of values for an *n* bit two's complement number is from 2^{n-1}-1 to -2^{n-1}. For a 4 bit number, the range would be from 2^3-1 to -2^3, or from 7 to -8.
- If you add one to the largest positive number, you get the largest negative number. For example, the largest 4 bit positive number is

[1] `https://en.wikipedia.org/wiki/Two%27s_complement`
[2] `https://en.wikipedia.org/wiki/Ones%27_complement`

`0111`. Adding one to this number gives `1000`, which is -8. Similarly, if you subtract one from the small negative number, you get the largest positive number. For example, the smallest 4 bit negative number is `0111`. Figure 3 shows XKCD's illustration of this property.

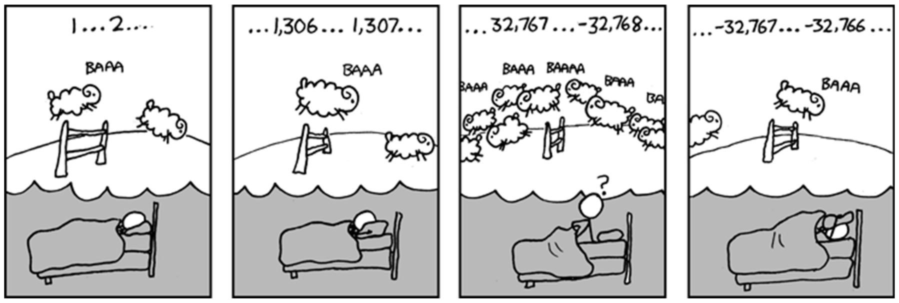


Figure 3: XKCD's Take on Integer Overflow
(used by permission of the artist)

Unsigned numbers have no sign bit, so all numbers are non-negative. The range of values for an *n* bit signed integer is from 0 to 2^n-1. For a 4 bit number, that means from 0 to 15. Adding one to the largest number, `1111`, results in overflow, with the result wrapping to zero, `0000`.

10.3 Integer Calculation Errors

The behavior of fixed-size integer arithmetic and conversions are subtle and, surprisingly, can easily lead to security vulnerabilities. Some of the most common types of flaws to be vigilant of are listed below. Given the range of languages and complexity of code, other variants are also possible.

Truncation: Some programming languages silently truncate the value during an assignment operation. This truncation happens as a result of type conversion between numeric values of different sizes. Since truncation of integers does not trigger an exception on most CPU architectures, this behavior is not surprising. Some languages may have ways to detect some of these problems either at runtime or with compilation options.

Overflow: Similar to truncation, some programming languages silently overflow the results of a numeric calculation. This is a subtle issue because overflow may be a desired behavior in some calculations, such as summing values for a checksum operation, and an undesired behavior in other cases, such as when calculating an array subscript.

Signed and Unsigned Arithmetic: Since the range that can be represented for a given size integer differs for signed and unsigned values, this introduces an additional set of problems. When combined with truncation or overflow, incorrect handling of signs can generate even more unexpected results.

Characters as Integers: In languages like C and its descendants, the `char` type is an integer, typically 8 bits long. Annoyingly, characters may or may not be signed integer, depending on the whim the processor architect and the compiler writer. The ISO C standard[3] is silent on this issue. Arithmetic operations on characters (such as summing them for a checksum) can be deceptively tricky, since `char` may be signed under one compiler or architecture and unsigned on another.

Assignment from Floating Point: The range of floating point numbers may exceed that of integer types so there is potential for overflow when assigning a floating point number to an integer. And, of course, floating numbers can represent non-integer values, so there can also be truncation or rounding errors when on such an assignment.

Intermediate Results: In a complicated computation, even though the final value of a computation is within range of the target type, overflow may occur for intermediate values at any step of a computation.

Computation with integers (including characters as integers) pervades modern computing in many forms. As discussed above, simple converting (casting) integer types can introduce errors. Computation with integers is always subject to these kinds of problems, and that includes not just arithmetic but also comparison and shifting.

Vigilance for numeric errors is required not just for math formulas, but also array indexes, buffer offsets, and other places that computation happens.

Many of these flaws only occur with extremely large or small numbers, or in corner cases. So vulnerable code will work nearly all the time, with hidden error cases that are rare and are difficult to detect without careful code review and testing. Each time that you port the code to a new platform, you have to recheck it.

Be aware that different compilers can potentially introduce flaws when the language specification does not precisely proscribe semantics. For example, the C language specifies minimum sizes for types; if a compiler uses a larger size, some computations may yield different results. In C and C++, since the `char` type can be either signed or unsigned, it is good practice to explicitly declare variables as signed or unsigned. To protect code that may be sensitive

[3] The most recent version of this standard can be found at `https://www.iso.org/standard/82075.html`. Be on the lookout for newer versions as they are released.

to the whims of the compiler where the language specification is lax, it is important to create test cases to ensure code that will not be broken by a compiler change.

10.4 Exploit Examples

Now, we will example a couple of examples where careless use of integers caused security vulnerabilities. These examples are based on vulnerabilities that we found in real systems.

10.4.1 Truncation Leading to Root (Administrator) Privileges

Consider this example function that executes a named program (prog), with arguments (argv), restricted to access a given working directory (jailDir), running the program with the specified user ID (uid) and group ID (gid). As a security precaution, the function checks that user and group IDs are not zero, to prevent starting the program running with root or administrator privileges.

```
void ExecUid(int uid, int gid,
             char *jailDir, char *prog,
             char *const argv[])
{
   if (uid == 0 || gid == 0) {
      /* On error, exit. So function does not return */
      FailExit("ExecUid: root uid or gid not allowed");
   }
   chroot(jailDir); /* restrict access to this dir */
   setuid(uid);     /* drop privileges */
   setgid(gid);
   execvp(prog, argv);
 }
```

Figure 4: Example of Truncation Causing Privilege Escalation

The problem with this code is that it incorrect specifies the types of the uid and gid arguments as int instead of uid_t and gid_t, respectively. Most parameters to system calls use specific defined types (such as uid_t) that specify the size of number and whether it is signed. For example, on the system that we evaluated this code, uid_t and gid_t were both 16 bits.

On this system, the compiler used 32 bit integers. So, when uid and gid are passed as parameters to setuid and setgid, they are recast as 16 bit numbers, resulting in truncation, the same as shown in Figure 1.

If the caller of this function specifies the uid and gid values to be 65536 (0x00010000) or any multiple of that value, they would pass the nonzero test in the if statement, but truncated to 16 bits when calling setgid and setuid, resulting in a value of zero being passed to the system calls. As a result, the user and group IDs of new program are root, in other words a privilege escalation.

10.4.2 Overflow Leading to Unauthorized Access to Memory

Integer overflow can also provide access to memory that should not be allowed. We saw an example of this behavior in OpenSSH versions 2.99 through 3.3. OpenSSH is a widely used open source utility that is critical to securely providing remote access to a system. Figure 5 shows the vulnerable code.

```
nresp = packet_get_int();
if (nresp > 0) {
  response = xmalloc(nresp*sizeof(char*));
  for (i = 0; i < nresp; i++)
    response[i] = packet_get_string(NULL);
}
```

Figure 5: Code in OpenSSH Vulnerable to Integer Overflow

The overflow bug occurs when the user input message causes the `packet_get_int` function to return an unexpectedly larger number and assign it to `nresp`. In this case, the number is 2^{30} (1,073,741,824 or 0x40000000).

The multiplication of `nresp*sizeof(char*)` results in $2^{30} \times 4$, which overflows the 32 bit integer resulting in truncation and a value of zero.

The zero value is passed to `xmalloc`, which is a wrapper function that calls `malloc` with the passed value. Note that calling malloc with a value of zero has unpredictable results that can vary from platform to platform. Sometimes it causes a zero to be returned, which should never be used as a pointer since, in most systems, it is not a valid memory address. Sometimes it returns a pointer to a valid memory address, but since no memory was allocated, the pointer should never be used because using it could overwrite other allocated memory.

The last assignment statement in the code assigns data that was sent by the user to the unauthorized area of memory. This vulnerability, combined with another one found in the same version of OpenSSH, allowed for arbitrary code execution.

10.5 More Integer Danger: Converting Strings to Integers

A common operation in programs is converting a number in character string format to a numeric type. This type of conversion has several possible pitfalls, especially when using C library functions that have little or no error checking.

For example, `atoi`, `atol`, and the `scanf` family – using `%u`, `%i`, `%d`, `%x`, and `%o` specifiers – do not detect overflow and values outside the range of representable integers. produce undefined results

In addition, if there are no valid numeric characters in the string passed to these functions, the conversion fails and results in zero value being returned.

In both the overflow and invalid conversion cases, there is no way for these functions to indicate that an error occurred. In general, these functions are dangerous to use, so you should write your own conversion function with careful error checking and reporting.

10.6 6 C# Examples

As we saw with memory checking, where the C# `unsafe` keyword can disable memory safety checks (Chapter 9), the C# language allows integer overflow checking to be controlled by the `unchecked` keyword. Checked code raises an exception in the case of integer overflow. Reasons for using unchecked code might be the case when the overflow is anticipated by the programmer and the truncated result is explicitly desired. Such a case can occur in simple checksum code.

Figure 6 shows an examples of a multiply operation, with and without checking. Executing the `UncheckedMethod` function in Figure *6* produces the somewhat surprising output:

```
Unchecked output value: -2
```

The explanation for this is easiest to understand by considering the hexadecimal values. The result of multiplying `0x7fffffff` by 2 is `0xfffffffe`, which is two's complement -2. This can be seen by adding 2 to the result to get `0x100000000`, which overflows the eight hex digits of a 32 bit integer and becomes zero.

Executing the `CheckedMethod` function, passing the maximum integer value (`0x7fffffff`) as a parameter, raises an exception due to an integer overflow. The `checked` keyword is applied to the multiplication operation so the compiler detects the overflow and control passes to the catch statement. The output from this exception appears in Figure 7.

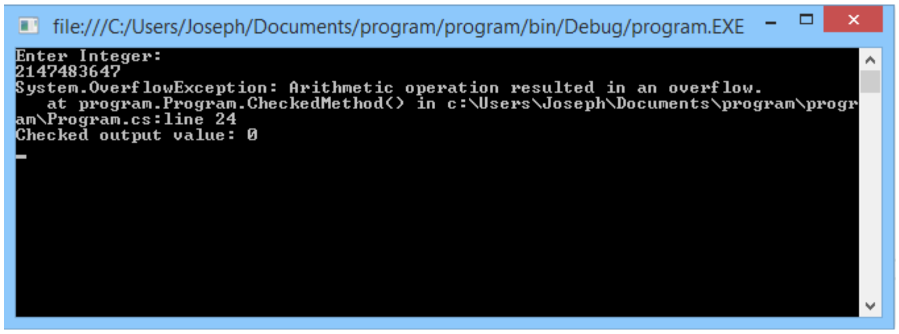
10.7 Numeric Overflow Mitigations

Many of the problems with integers result from sloppy use of types, so the first step to writing more secure code is to take meticulous care with types.

Notice any conversions, which are often implicit, mixing of signed and unsigned types, and especially down-casting into smaller size types.

```
static void Main(string[] args) {
  UncheckedMethod (Int32.MaxValue);
  CheckedMethod (Int32.MaxValue);
}

static void UncheckedMethod(int x) {
  const int y = 2;
  int z=0;

  unchecked {
    z = x * y;
  }
  Console.WriteLine(“Unchecked output value: {0}”, z);
}

static void CheckedMethod (int x) {
    const int y = 2;
    int z=0;

    try {
       z = checked (x * y);
    }
   catch (System.OverflowException e) {
       Console.WriteLine(e.ToString());
    }
    Console.WriteLine(“Checked output value: {0}”, z);
}
```

Figure 6: C# Arithmetic Operations without and with Overflow Checking


file:///C:/Users/Joseph/Documents/program/program/bin/Debug/program.EXE

```
Enter Integer:
2147483647
System.OverflowException: Arithmetic operation resulted in an overflow.
   at program.Program.CheckedMethod() in c:\Users\Joseph\Documents\program\progr
am\Program.cs:line 24
Checked output value: 0
```

Figure 7: Output from the Exception in the CheckedMethod Function from Figure 6

You should always validate the range of values to be reasonable before doing computation that can lead to overflow to ensure that the largest reasonable values are always going to be within range of what the type can represent. With signed types, you should also consider the most negative values as well.

When necessary, add code to check for overflow or use safe integer libraries or large integer libraries. If the compiler does not provide a facility to check for overflow, it can be done in your code but requires careful attention to detail. For example, you can detect overflow in C if the sum of two unsigned values is smaller than the numbers being summed.

Avoid mixing signed and unsigned integers in a computation. The easiest solution is to cast these to a larger signed type that can represent the full range of both values.

Conversions involving floating point and other numeric representations can also produce similar problems.

When available, use compiler options for integer overflow warnings and runtime exceptions, including handling the exceptions so an overflow does not terminate the process and become a potential denial of service.

For C/C++ when converting strings to numeric values, use `strtol`, `strtoul`, `strtoll`, and `strtoull`, which provide error detection.

10.8 The Cost is Great

The cost of not checking your integer operations can have enormous consequences. A dramatic example happened back in June of 1996, with millions of television viewers worldwide watching the consequences. In this case, it was a 64 bit floating point number assigned to a 16 bit integer, where there were no checks. In this particular case, an overflow occurred leading to a failure. Any guesses on what was the event?

It was the first launch of the Ariane 5 space vehicle. Ariane 5 was the successor to the highly successful Ariane 4, with the new rocket being more powerful. The added power allowed it to launch larger payloads into orbit, and, importantly, reach a wider range of orbits, notably those further off the equator. And that is where the problem came in. Ariane 5 used a large amount of software that was already certified on Ariane 4. Unfortunately, somewhere in that flight control software was a floating-point number that tracked latitude, and was later assigned to a 16 bit integer.

The first mission launched Ariane 5 to a higher latitude than previously used in Ariane 4, causing an overflow on that integer. The primary flight control computer crashed and, a moment later, the backup computer (running the same software) also crashed. The rocket became a large, fast moving, and explosive object that could not be steered. As it was supposed to in this situation, it self-destructed less than a minute after launch.

Figure 8: Ariane 5 Mission 501 (First Launch)

And it was an expensive overflow. The cost of the lost vehicle was around $500 million and the total cost to fix the design issues that caused this problem, including the years of delay, cost around $7 billion. Do not let this happen to you!

10.9 Summary

It was the first launch of the Ariane 5 space vehicle. Ariane 5 was the successor to the highly successful Ariane 4, with the new rocket being more powerful. The added power allowed it to launch larger payloads into orbit, and, importantly, reach a wider range of orbits, notably those further off the equator. And that is where the problem came in. Ariane 5 used a large amount of software that was already certified on Ariane 4. Unfortunately, somewhere in that flight control software was a floating-point number that tracked altitude, and was later assigned to a 16 bit integer.

10.10 Exercises

For each of these exercises, you have free choice of what programming language to use. It is even more interesting to try each of these exercises in more than one language.

1. Write code to do a simple calculation with fixed size integers, then test it with different input values to observe what happens when there is overflow.

2. What is the potential error condition when doing a multiply or divide by `-1` when using two's complement integers?
3. Write functions that do 32 bit signed integer arithmetic returning a correct 32 bit signed integer result or throw an exception when overflow occurs. For each function, do not use compiler overflow checking, and do not up-cast to a larger capacity type. Also, for each function, try to specify what are the tricky corner cases that might cause your function to misbehave.
    a. (Easy) Write a function, `add(x,y)` that returns the 32 bit sum of two 32 bit signed integers or in the case of overflow raises an exception.
    b. (Slightly harder) Copy and adapt the `add` function to do `subtract(x,y)`, the difference of the two integers, detecting overflow as above.
    c. (Harder) Write `multiply(x,y)`, the product of the two integers, again, detecting overflow. Why is this harder than doing the `add` or `subtract`?
    d. (Harder, or is it?) Similarly, write `divide(x,y)`, which divides two integers, `x` by `y`. Division by zero is undefined, but this is a separate exception to overflow. Is overflow detection necessary? Why or why not?
4. Write a numeric parsing function that converts a character string of decimal digits to an unsigned 32 bit integer value, or raise an exception if the value exceeds the maximum possible value.
5. In Figure 4, the call to `setuid` appears before the call to `setgid`. Why is the order problematic and why would reversing the order fix the problem?