Introduction to Software Security

Chapter 3.2: Numeric Errors

Loren Kohnfelder
loren.kohnfelder@gmail.com

Elisa Heymann
elisa@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

DRAFT Revision 2.0, January 2022.

1 Objectives

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

Examples from C/C++ and C#.

1.1 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 commonly 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.

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

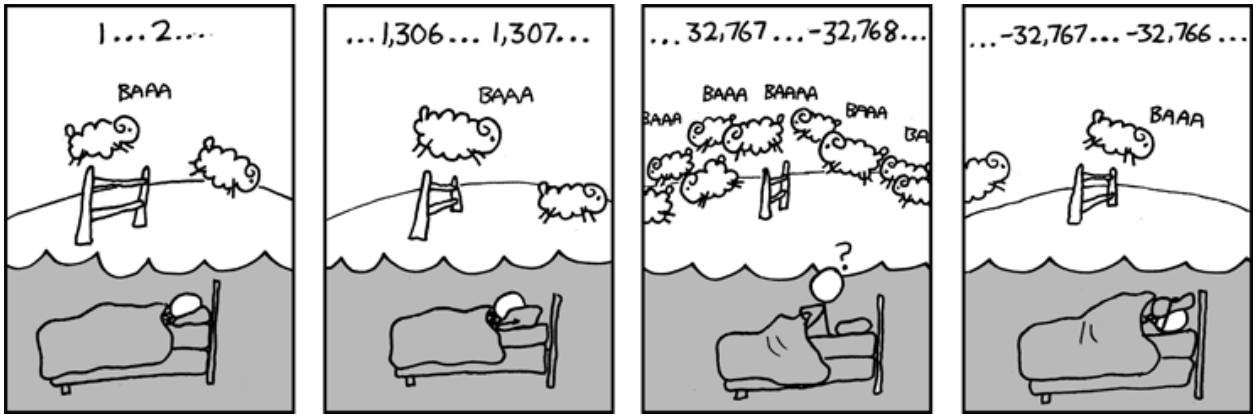
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

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

0	0	0	0	0	0	0	0

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.

Since signed integers are represented as two's-complement, even more surprising results are possible. The details of two's-complement are outside our scope here, but to explain briefly it uses the properties of modular arithmetic so that for a constrained range of values and operations the same integer arithmetic hardware can be used for either signed or unsigned computation. For example, a signed 8-bit integer represents values between -128 and +127.



https://xkcd.com/571/

3 Integer calculation errors

The peculiarities of fixed-size integer arithmetic and conversions are subtle and can easily lead to serious security vulnerabilities. Some of the most common kinds 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. 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 and it may or may not be signed so operations with characters can be deceptively tricky.
- Floating point: The range and precision of floating point numbers may exceed that of integer types but still there are limitations to accuracy as well as truncation or rounding errors when the result is converted to an integer value. Very small values may be subject to underflow as well.
- Intermediate results: 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 (often 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 many 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. Also in C, the char type can be either signed or unsigned, so it is good practice to explicitly declare variables as signed or unsigned. To protect code that may be sensitive 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.

4 Exploit Examples

Consider this example routine that executes a named command (prog) with arguments (argv) in the context of a given working directory (jailDir) running as a given user (uid) of a group (gid). As a security precaution, the programmer checks that zero IDs (which specify the root superuser) are disallowed.

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

```
  setuid(uid); /* drop privs */
  setgid(gid);
  execvp(prog, argv);
}
```

The problem with this code is incorrectly specifying the types of the uid and gid arguments as int instead of uid_t and gid_t, respectively. If the operating system uses 16-bit IDs, but compiler uses 32-bit integers, then uid and gid values of 65536 (or any multiple of that number) would pass the nonzero test, but setgid and setuid would be called with values that would be cast to 0.

And problems like this happen in real applications as well. For example, consider this is classic overflow from OpenSSH 3.3 (this is a very high profile and widely used open source utility that obviously is critical to securely afford remote access).

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

From https://www.owasp.org

The overflow bug occurs when the multiplication computing the size in bytes of the response buffer exceeds the range of values an integer holds. For example, for a 32 bit compiler, consider when nresp has the value 2^{30} (1,073,741,824 or in hexadecimal 0x40000000): for a pointer, sizeof(char*) is 4 so the product nresp * sizeof(char*) is 2^{32} (0x100000000) which overflows, yielding a zero argument to xmalloc() which raises an error. The if statement is intended to protect against passing zero to xmalloc because the overflow produces a zero value the error still can happen. Also note that a value such as 2^{30} + 1 would overflow and end up creating a small 4 byte buffer that the following for loop would exceed, resulting in a buffer overflow.

5 Integer Danger Signs

Integer overflow bugs can be subtle before the code works correctly for a large range of normal values. Here are some of the most common signs to look for that suggest faulty code:

- Not checking for overflow.
- Mixing integer types of different sizes.
- Mixing unsigned and signed integers.
- Mixing system defined types and standard compiler types.
- Conversion to integer type of smaller size.

In some languages there can be subtle differences between an API typedef type and a built-in type. For example, in C the size of an array (results of sizeof) is of unsigned type size_t but the difference of two pointers that point to the end and beginning of that array is ptrdiff_t, which is signed.

These danger signs may not always be explicit and are easily overlooked. One common case is in a function call where argument passing implicitly applies type conversions according to how the callee declares its parameters. Also note that overflows may occur in the intermediate results of an expression that might look safe.

An example will clarify this last point: the expression INT_MAX*2/3 might appear to compute ⅔ of the maximum signed integer value, but since the multiplication by two happens first that causes an overflow before the divide happens. Instead, by writing (INT_MAX/3)*2 and doing the division first avoids overflow (since the result is always of lesser magnitude) but of course yields subtly different results because of rounding.

Numeric parsing is a common source of numeric overflows because the class standard functions — atoi, atol, atof, 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. Also be aware that conversions of text that is not a valid numeric string results in zero.

6 C# Examples

The C# language allows integer overflow checking to be controlled by the code or by compiler option. Checked code raises an exception in the case of integer overflow. Reasons for using unchecked code might be for maximum speed of computation (in a critical section executed many times), or in the rare case when the overflow is anticipated by the programmer and the truncated result is explicitly desired.

The following examples illustrate both without and with checking as specified in the code.

```
static void Main(string[] args) {
  UnCheckedMethod (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);
}
```

Derived from http://msdn.microsoft.com/us-en/library/a569z7k8%28v=vs.90%29.aspx

Executing this code above 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 0).

The next example explicitly specifies overflow checking for the multiplication.

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

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);
}
```

Running this code with the maximum integer value for x (`0x7fffffff`) as above, this time 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.

7 Numeric Overflow Mitigations

Many of the problems with integer 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.

Validate the range of values to be reasonable before doing computation that can lead to overflow, and ensure that the largest reasonable values are always going to be within range of what the type can represent. With signed types 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 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.

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 Denial of Service exposure.

For C/C++ parsing of numbers, use `strtol`, `strtoul`, `strtoll`, `strtoull`, `strtof`, `strtod`, `strtold` which provide error detection.

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?

Ariane 5 Mission 501 (First Launch)

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's 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.

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 couldn't be steered. As it was supposed to in this situation, it self-destructed less than a minute after launch.

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

9 Summary

Software is full of integer computations which seem simple but there are a number of issues that can arise and easily lead to serious security vulnerabilities.

- Anticipate overflow for any integer computation.
- Only use mixed types (both size and signed/unsigned) with care.
- Range check values before any operation that might overflow.
- Use compiler facilities to detect overflow and conversion errors.
- Use safe integer libraries that detect overflow and invalid cases.

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. 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 sum or difference?
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?

3. 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.