A Guide to Undefined Behavior in C and C++, Part 2

Also see Part 1 and Part 3.

When tools like the bounds checking GCC, Purify, Valgrind, etc. first showed up, it was interesting to run a random UNIX utility under them. The output of the checker showed that these utility programs, despite working perfectly well, executed a ton of memory safety errors such as use of uninitialized data, accesses beyond the ends of arrays, etc. Just running grep or whatever would cause tens or hundreds of these errors to happen.

What was going on? Basically, incidental properties of the C/UNIX execution environment caused these errors to (often) be benign. For example, blocks returned by malloc() generally contain some padding before and/or after; the padding can soak up out-of-bounds stores, as long as they aren't too far outside the allocated area. Was it worth eliminating these bugs? Sure. First, an execution environment with different properties, such as a malloc() for an embedded system that happens to provide less padding, could turn benign near-miss array writes into nasty heap corruption bugs. Second, the same benign bugs could probably, under different circumstances, cause a crash or corruption error even in the same execution environment. Developers generally find these kinds of arguments to be compelling and these days most UNIX programs are relatively Valgrind-clean.

Tools for finding integer undefined behaviors are less well-developed than are memory-unsafety checkers. Bad integer behaviors in C and C++ include signed overflow, divide by zero, shift-past-bitwidth, etc. These have become a more serious problem in recent years because:

- Integer flaws are a source of serious security problems
- C compilers have become considerably more aggressive in their exploitation of integer undefined behaviors to generate efficient code

Recently my student Peng Li implemented a checking tool for integer undefined behaviors. Using it, we have found that many programs contain these bugs. For example, more than half of the SPECINT2006 benchmarks execute integer undefined behaviors of one kind or another. In many ways the situation for integer bugs today seems similar to the situation for memory bugs around 1995. Just to be clear, integer checking tools do exist, but they do not seem to be in very widespread use and also a number of them operate on binaries, which is too late. You have to look at the source code before the compiler has had a chance to exploit — and thus eliminate — operations with undefined behavior.

The rest of this post explores a few integer undefined behaviors that we found in LLVM: a medium-sized (~800 KLOC) open source C++ code base. Of course I'm not picking on LLVM here: it's very high-quality code. The idea is that by looking at some problems that were lurking undetected in this well-tested code, we can hopefully learn how to avoid writing these bugs in the future.

As a random note, if we consider the LLVM code to be C++0x rather than C++98, then a large number of additional shift-related undefined behaviors appear. I'll talk about the new shift restrictions (which are identical to those in C99) in a subsequent post here.

I've cleaned up the tool output slightly to improve readability.

Integer Overflow #1

Error message:

```
UNDEFINED at <BitcodeWriter.cpp, (740:29)> :
Operator: -
Reason: Signed Subtraction Overflow
left (int64): 0
right (int64): -9223372036854775808
```

Code:

```
int64_t V = IV->getSExtValue();
if (V >= 0)
  Record.push_back(V << 1);
else
  Record.push_back((-V << 1) | 1);   <<----- bad line
```

In all modern C/C++ variants running on two's complement machines, negating an int whose value is INT_MIN (or in this case, INT64_MIN) is undefined behavior. The fix is to add an explicit check for this case.

Do compilers take advantage of this undefined behavior? They do:

```
[regehr@gamow ~]$ cat negate.c
int foo (int x) __attribute__ ((noinline));
int foo (int x)
{
  if (x < 0) x = -x;
  return x >= 0;
}

#include <limits.h>
#include <stdio.h>

int main (void)
{
  printf ("%d\n", -INT_MIN);
  printf ("%d\n", foo(INT_MIN));
  return 0;
}
[regehr@gamow ~]$ gcc -O2 negate.c -o negate
negate.c: In function `main':
negate.c:13:19: warning: integer overflow in expression [-Woverflow]
[regehr@gamow ~]$ ./negate
-2147483648
1
```

In C compiler doublethink, -INT_MIN is both negative and non-negative. If the first true AI is coded in C or C++, I expect it to immediately deduce that freedom is slavery, love is hate, and peace is war.

Integer Overflow #2

Error message:

```
UNDEFINED at <InitPreprocessor.cpp, (173:39)> :
Operator: -
Reason: Signed Subtraction Overflow
left (int64): -9223372036854775808
right (int64): 1
```

Code:

```
MaxVal = (1LL << (TypeWidth â€“ 1)) â€“ 1;
```

In C/C++ it is illegal to compute the maximum signed integer value like this. There are better ways, such as creating a vector of all 1s and then clearing the high order bit.

Integer Overflow #3

Error message:

```
UNDEFINED at <TargetData.cpp,  (629:28)> :
Operator: *
Reason: Signed Multiplication  Overflow
left (int64): 142998016075267841
right (int64): 129
```

Code:

```
Result += arrayIdx * (int64_t)getTypeAllocSize(Ty);
```

Here the allocated size is plausible but the array index is way out of bounds for any conceivable array.

Shift Past Bitwidth #1

Error message:

```
UNDEFINED at <InstCombineCalls.cpp, (105:23)> :
Operator: <<
Reason: Unsigned Left Shift Error: Right operand is negative or is  greater than or equal to the width of the p
```

```
left (uint32): 1
right (uint32): 63
```

Code:

```
unsigned Align = 1u << std::min(BitWidth â€ “ 1, TrailZ);
```

This is just an outright bug: BitWidth is set to 64 but should have been 32.

Shift Past Bitwidth #2

Error message:

```
UNDEFINED at <Instructions.h, (233:15)> :
Operator: <<
Reason: Signed Left Shift Error: Right operand is negative or is greater  than or equal to the width of the pro
left (int32): 1
right (int32): 32
```

Code:

```
return (1 << (getSubclassDataFromInstruction() >> 1))  >> 1;
```

When getSubclassDataFromInstruction() returns a value in the range 128-131, the right argument to the left shift evaluates to 32. Shifting (in either direction) by the bitwidth or higher is an error, and so this function requires that getSubclassDataFromInstruction() returns a value not larger than 127.

Summary

It is basically evil to make certain program actions wrong, but to not give developers any way to tell whether or not their code performs these actions and, if so, where. One of C’s design points was “trust the programmer.” This is fine, but there’s trust and then there’s trust. I mean, I trust my 5 year old but I still don’t let him cross a busy street by himself. Creating a large piece of safety-critical or security-critical code in C or C++ is the programming equivalent of crossing an 8-lane freeway blindfolded.

July 23, 2010 regehr Computer Science, Software Correctness

---

7 responses to “A Guide to Undefined Behavior in C and C++, Part 2”

1. **Neil Conway** says:
   July 23, 2010 at 3:03 pm

   The analysis tool for undefined integer operations sounds very interesting — are there any plans to release it publicly?

2. **regehr** says:
   July 23, 2010 at 4:58 pm

   Hi Neil– Yes, definitely! Not sure when, but hopefully in the next month or two.

3. **Eric Eide** says:
   July 23, 2010 at 10:06 pm

I'm pretty sure that war is peace, not peace is war. I'm pretty sure that in doublespeak, "is" is double-plus unsymmetric.

4. **Ben L. Titzer** says:
   July 24, 2010 at 9:00 am

   Old Russian proverb: "trust by verify".

   Ugh, those pesky shifts. I recently altered the Virgil specification to mandate that shifts larger than the bitwidth of the type produce zero, as if all the bits were shifted out. Java chose differently: only the lower 5 bits of the shift value are used for int shift (6 for a long shift). This typically means generating a branch for most targets. But most shift amounts are constants and the check can be eliminated. I'm crossing my fingers that this won't be a big deal in the future.

5. **regehr** says:
   July 24, 2010 at 9:55 am

   Ben– I think your approach is definitely better. I didn't know Java masked off the higher bits, that's not semantically clean.

6. **sh** says:
   October 13, 2010 at 10:13 am

   "In C/C++ it is illegal to compute the maximum signed integer value like this. There are better ways, such as creating a vector of all 1s and then clearing the high order bit."

   According to C99 6.2.6.2 it's undefined whether this gives you a valid integer value. An implementation might choose to represent signed integers using padding bits in such a way that this approach creates a trap value.

7. **regehr** says:
   October 13, 2010 at 10:24 am

Hi Sebastian- The value of padding bits is unspecified (not undefined) and the presence or absence of padding bits is implementation defined. Of course I could have been more precise and said “There are better ways on C implementations whose implementation-defined behavior for integer representations is conventional…” But this didn’t seem to need to be said….