Embedded in Academia

{ 2013 11 13 }

Integer Undefined Behaviors in Open Source Crypto Libraries

Crypto libraries should be beyond reproach. This post investigates integer-related undefined behaviors found in crypto code written in C/C++. Undefined behavior (UB) is bad because according to the standards, it destroys the meaning of any program that executes it. In practice, over the last decade compilers have become pretty smart about exploiting integer undefined behaviors to generate fast but surprising object code. Plenty of examples are here.

This post was motivated by a discussion on the C++ standards committee's mailing list for undefined behavior where I proposed making signed left-shift work just like unsigned left-shift. In contrast, in C99, C11, and C++11, it is illegal to shift a 1 bit into, out of, or through the sign bit. Many developers are unaware of this restriction. This seemed to me like a pretty safe proposal since it isn't clear that any existing compiler implements anything other than two's complement semantics for signed left shifts in the first place — so the change would essentially just mandate the behavior that is already implemented. Chandler Carruth from Google disagreed with this, stating that not only do the LLVM people believe that these UBs can result in useful optimizations but also that developers were appreciative of the bug reports that come from statically and dynamically detecting signed left-shift problems. Of course, if the UBs disappear due to a tighter language standard, then the capability for error detection disappears too. Chandler's messages contain some good details so I'll just link to them: 1, 2, 3. Based on that conversation, my goal here is to try to understand where these integer undefined behaviors come from and whether they correspond to "real bugs" or not.

To make matters just a little more complicated, there's a C++ defect report stating that it should be permissible to shift a 1 into the sign bit, but not out of or through it. The defect report has no effect upon any of the standards until a TC is released, but it does indicate that this particular issue is considered to have been fixed by the committee. There is no corresponding defect report for C, as far as I know.

Let's get on with the testing. The methodology used here was, for each of a number of open source crypto libraries that include a substantial amount of C/C++ code:

1. Grab a recent version
2. Build it using Clang with the `-fsanitize=integer` flag on an x86-64 machine running Linux
3. Run the library's built-in test suite
4. Examine the resulting undefined behaviors

Note that if you do this kind of work yourself, you're most likely better off using the `-fsanitize=undefined` flag because it doesn't warn about (well-defined) unsigned overflows and it warns about many undefined behaviors not relating to integers.

BeeCrypt 4.2.1

This library executes no integer undefined behaviors while running its test suite. Nice!

Botan 1.11.4

Two of Botan's rotate functions, found in `rotate.h`, take an unsigned int and shift it by 32 places. The code looks like this:

```
1  template inline T rotate_left(T input, size_t rot)
2  {
3    return static_cast((input << rot) | (input >> (8*sizeof(T)-rot)));;
4  }
```

The problem occurs not when rotating by 32, but rather when rotating by 0. Rotate by 0 is undefined because it causes the second shift operator to shift by 32 – 0. The fix (applied by the code's author already) is to add a check for rotate-by-0 and in that case return the unmodified input. This is a little bit annoying because in the expected case where the rotate code is translated into a rotate instruction (both GCC and Clang will do this) the test just adds overhead to the generated code since the machine instruction is perfectly well defined for rotating by zero. The compiler could go ahead and remove the test and exit, but neither has been taught to do so.

Shifting by bitwidth (or more) is a serious kind of undefined behavior that can lead to portability problems across compilers, platforms, and even optimization levels. On the version of Clang from Xcode 5, a little shift-past-bitwidth program that I wrote has different results each time it is run.

HELib from Github on 10/26/13

This library executes numerous integer undefined behaviors, but all of them occur inside code from NTL. However, NTL can be configured using the `NTL_CLEAN_INT` option, which makes the undefined behaviors go away. In other words, the NTL author (who I had a short discussion with) is aware of the undefined behaviors but prefers the version that executes them since it results in faster code and compilers do not currently (as far as we know) exploit the undefined behaviors to break NTL. My own take is that if NTL were distributed as a library that could be solidly unit tested, this might not be a bad bet to make. However, a lot of NTL code gets included in applications via header files, giving the compiler plenty of opportunity to do tricky things. Personally, I would not trust a modern compiler to fail to exploit an undefined integer overflow when it can see all of the code and propagate constants through it.

I didn't report anything to the HELib people since the UBs aren't in their code, and their code appears to be intended for research purposes anyhow.

LibTomCrypt 1.17

This library contains an interesting undefined behavior in a line of code (anubis.c:934) that shifts a value left by 24:

```
1  for (i = 0, pos = 0; i < N; i++, pos += 4) {
2    kappa[i] =
3       (key[pos    ] << 24) ^
4       (key[pos + 1] << 16) ^
5       (key[pos + 2] <<  8) ^
6       (key[pos + 3]      );
7  }
```

No undefined behaviors can occur when shifting an unsigned value left by 24. However, in this code `key` is a pointer to unsigned char. The char value is promoted to int before being shifted; if the value in the int is >127, a 1 bit will end up being shifted into the sign bit, which is undefined. This example nicely illustrates how the integer promotion rules in C/C++ can have surprising consequences. There's another

instance of the same problem in the same file at line 1051. I reported these issues but haven't heard back from the developers yet.

Libgcrypt 1.5.3

Ok, at this point things start to get a bit repetitive so I'll start leaving out the details. This version of Libgcrypt contains:

- 4 locations in blowfish.c, 8 locations in cast5.c, 3 locations in des.c, and 8 locations in twofish.c where an unsigned char is promoted to int and then left-shifted by 24 places, resulting in undefined behavior
- 2 locations in cast5.c where a 32-bit integer is shifted by 32 places, again as part of a rotate operation

Keep in mind that the former problems are probably benign for now, whereas the latter ones should definitely be fixed. These have been reported (see developer's response here).

Crypto++ 5.6.2

3 locations in misc.h where a 32-bit unsigned value is shifted by 32 places, again in rotate functions. Reported.

Sodium 0.4.5

3 locations in aes256-ctr.c where an unsigned char is promoted to signed int and shifted left by 24. Reported, and fixed within hours.

Libmcrypt 2.5.8

Summary:

- 2 locations in cast-128.c where the rotate operation leads to a shift by 32 places of a 32-bit integer
- 26 locations in cast-256.c with very large shift exponents up to almost 2^{32}; the code is heavily macroized so I didn't dig in too deeply
- 2 signed multiplication overflows in enigma.c
- 1 location in des.c, 1 in gost.c, and 2 in loki97.c where a 1 is left-shifted into the sign bit
- 1 location in tripledes.c where a 1 is left-shifted out of the sign bit

I didn't bother reporting since libmcrypt was last released in 2007 and it has a number of long-open, serious-looking bugs. Hopefully most people have phased out their use of this library, though I see that it is still provided as a package in Ubuntu 13.10.

Nettle 2.7.1

1 location in blowfish.c and 1 in twofish.c where an unsigned char is promoted to signed int and the left-shifted by 24. 2 locations in cast128.c where a 32-bit value is shifted by 32 places in a rotate function.

Reported and the rotate has been fixed already.

OpenSSL SNAP-20131112

At a_int.c:397 and obj_dat.c:143 there are undefined signed left-shifts. These — unlike most of what we've seen so far — are the result of variables that were declared as signed, as opposed to being the result of implicit conversions. These shifts should be done more carefully, or should be done using unsigned operations.

In c_enc.c there are 20 sites that shift a 32-bit variable by 32 places, all resulting from a macro called E_CAST.

gost89.c has 8 locations where a 1 is left-shifted into the sign bit, all are due to promotion of an unsigned char to int. Same thing happens at 2 locations in gost_crypt.c.

These have been reported.

The Bad News

The bad news, obviously, is that nearly all of the crypto libraries I tested execute integer undefined behaviors. There appear to be several root causes:

1. C's weird integer promotion rules practically guarantee that signed types will be introduced into computations that start out with only unsigned types. Signed math is hard because its operators have much stricter rules to follow if we are to avoid undefined behavior. Unsightly explicit casts to unsigned are the solution.
2. Some developers continue to purposefully use signed math under the faulty assumption that the C/C++ languages are built around two's complement arithmetic
3. Reasoning about function preconditions is hard even for experienced developers. In my opinion, some of these libraries could have used a lot more assertions to go along with their (generally perfectly adequate) test suites.

The Good News

1. Most of the undefined behaviors seen here stem from a few simple patterns, most notably "rotate by zero" and "unsigned char is promoted to signed int." Crypto developers can and should learn to recognize and deal correctly with these. In general, the problems seen here are not difficult to fix.
2. The developers who have responded to my bug reports seem very on the ball and generally happy that people are vetting their software.
3. Clang's undefined behavior sanitizer revealed all of these problems trivially. Use it, folks.

Next Steps

The undefined behaviors reported here are a subset of all undefined behaviors executed by these libraries, in three senses:

1. I only checked for integer undefined behaviors, whereas Clang's undefined behavior sanitizers can find other problems such as memory safety bugs.

2. Even Clang's full set of undefined behavior sanitizers checks only a subset of all undefined behaviors in C/C++. More work on checkers is needed if we are to keep using C/C++ in security-critical roles.
3. Even if we had a checker for all undefined behaviors, we would still be limited by the quality of each library's builtin test suite. Static analysis is one solution and better testing is another.

If our security relies on crypto libraries operating correctly, it would not be a bad idea for someone to spend some time and energy vetting crypto code using a tool like Frama-C. Crypto code is nice and mathy: a perfect target for formal methods.

Finally, if I've missed your favorite crypto library, please let me know and I'll take a look. I am interested only in open source code which contains a substantial amount of C/C++ and which can be built on Linux or OS X.

Posted by regehr on Wednesday, November 13, 2013, at 9:52 am. Filed under Compilers, Computer Science, Software Correctness. Follow any responses to this post with its comments RSS feed. Both comments and trackbacks are currently closed.

{ 13 } Comments

1. Phil | November 13, 2013 at 10:55 am | Permalink

   One library not included that jumps out at me is Mozilla's libnss, which is fairly popular, and often used as an alternative to OpenSSL due to its more liberal license terms.

2. Max Lybbert | November 13, 2013 at 5:33 pm | Permalink

   > if I've missed your favorite crypto library, please let me know and I'll take a look. I am interested only in open source code which contains a substantial amount of C/C++ and which can be built on Linux or OS X.

   I would like to see how NACL holds up (http://nacl.cr.yp.to/). It builds on Linux. I haven't tried OS X, but I'd expect that to work as well.

3. Max Lybbert | November 13, 2013 at 5:35 pm | Permalink

   And now I'd like to withdraw my comment: it looks like the analysis of sodium would have covered NaCl.

4. pm215 | November 14, 2013 at 6:18 am | Permalink

   My personal preference would be for the standards committee to mandate 2s-complement arithmetic, which would knock out a lot of these random "language doesn't behave the way most programmers think it does" UB bugs. The weird stuff like sign-magnitude 48 bit integers with padding bits can be relegated to an appendix describing which constraints are relaxed in those cases, and compilers should default to the tightened-up behaviour unless the programmer uses std=relaxedc or something.

In other words, specify and implement the language people are actually writing in practice, not a closely related one with a lot of extra beartraps for the unwary.

5. bcs | November 14, 2013 at 7:44 am | Permalink

   @pm215: and as a counter point, is allowing those UB to be defined (and this permit-able) a significant lost bug finding and/or optimization opportunity? If 99% of UB instances found in the real world (it's not likely *that* high) are in fact bugs, then I'd be tempted to say that it should go the other way and require the compile to reject code that encounters them.

6. pm215 | November 14, 2013 at 8:35 am | Permalink

   @bcs: yes, indeed, I'm happy for some of these cases to get defined as "always a compile time error". "Always a runtime error" is probably OK too. The nasty ones are "silently accepted but does something deeply unexpected at runtime" and "silently accepted and works fine on this compiler but may change to do something deeply unexpected on a newer compiler"; it's the latter that are giving compilers a bad reputation for performing "adversarial optimizations", where they take the interpretation of the C standard that is worst for the developer and best for the compiler's benchmark figures…

7. regehr | November 14, 2013 at 9:17 am | Permalink

   I used to take a pretty strong position similar to the one pm215 advocates, where erroneous program behavior must be trapped at compile time or run time. I've softened a bit now, and would happily settle for a dynamically safe systems programming language with optional unsafe compilation modes where safety checks are omitted. This would be an acceptable tradeoff for performance-critical code that has been extensively tested, formally verified, or whatever. It's vaguely possible that C/C++ will eventually make it there if for example Intel's bounds checking hardware ends up working well. More likely, however, is that newer systems languages like Rust, Go, D, and ones not yet invented end up occupying most of the niche that C/C++ occupy now.

8. regehr | November 14, 2013 at 9:23 am | Permalink

   The tradeoff that bcs mentions is a real one. For example, we might ask if Java programs are free of integer overflow bugs because its integers are two's complement? Of course they are not.

9. Andrey Karpov | November 14, 2013 at 11:18 pm | Permalink

   I am glad that it was brought to attention. I wrote about the shift operations, but no one hears.

   My article: Wade not in unknown waters. Part three – http://www.viva64.com/en/b/0142/

   My examples of errors in open-source projects: http://www.viva64.com/en/examples/V610/

10. Frank Denis | November 16, 2013 at 11:06 am | Permalink

    "Hopefully most people have phased out their use of [mcrypt]" > Unfortunately not, it's heavily used with PHP. In fact, most of the PHP applications doing crypto depend on it.

11. ch | November 17, 2013 at 8:48 am | Permalink

    PolarSSL is also an interesting library. Builds on Linux and likely OS X.

12. regehr | November 17, 2013 at 9:15 am | Permalink

ch, PolarSSL looks pretty clean: just two spots (lines 291 and 295 of camellia.c) where a 1 is shifted into the sign bit.

13. Mate Soos | November 18, 2013 at 3:02 am | Permalink

    I first heard about “-fsanitize=YYY” clang flags from your blog and I’ve since found lots of nice bugs with them. It’s really great and it has become a tool that I use regularly, just like valgrind.