Little CMS2

Interger overflow issue

https://www.littlecms.com

Introduction

This short article discusses an issue discovered during the AI Cyber Challenge (AIXCC), a remarkable competition where the Little CMS codebase was among the open-source repositories examined. Derek Zimmer from OSTIF reached out to me regarding a potential problem they had identified, and David Korczynski provided all the necessary details.

At first glance, the issue appeared to be a signed integer overflow; nothing more than the usual unpleasantness, with no serious implications beyond producing incorrect colors. It did not seem exploitable. However, when I attempted to fix it, I uncovered a rather intriguing and unfortunate coincidence that makes this bug worthy of a detailed write-up. I hope you find it interesting.

Issue description

The original report was about a signed integer overflow in following function:

```
cmsINLINE cmsS15Fixed16Number _cmsToFixedDomain(int a)
{
        return a + ((a + 0x7fff) / 0xffff);
}
```

The stack dump for this error was

```
lcms2_internal.h:151:85: runtime error: signed integer overflow: -2147450908 + -32767
cannot be represented in type 'int'
   #0 0x5592eb3423ab in _cmsToFixedDomain /src/lcms/src/./lcms2_internal.h:151:85
   #1 0x5592eb3423ab in Eval4Inputs /src/lcms/src/cmsintrp.c
```

At first glance, it seems to be a simple case of missing boundary checks, which could be resolved by adding appropriate guards. Something like the fix proposed fix by the bug's discoverer:

```
cmsINLINE cmsS15Fixed16Number _cmsToFixedDomain(int a)
{
   int64_t tmp = (int64_t)a + (((int64_t)a + 0x7fff) / 0xffff);
   if (tmp > INT32_MAX) tmp = INT32_MAX;
   if (tmp < INT32_MIN) tmp = INT32_MIN;
   return (cmsS15Fixed16Number)tmp;
}
```

Perhaps it is because this function is critical and invoked frequently. Or maybe it relates to the use of int64, which I cannot reasonably adopt. In any case, the suggested fix triggered a warning in my mind: something was not behaving as expected. Rather than being the root cause, this issue seemed more like a symptom pointing to the real origin of the problem.

Analysis

Image data, when encoded in 8 or 16 bits uses all available bits for storing levels. So, in 8 bits we go from 0 to 0xff and in 16 bits we go from 0 to 0xffff. Another way to encode channel numbers is by using floating point. In that case, a frequent representation is from 0.0 to 1.0. The representation of mantissa-exponent is not so efficient. Floating point is very convenient for math manipulation, but unfortunately it may be slow on some small platforms like MCUs and dsPICs. Since LittleCMS is widely used on those platforms for device firmware and IoT, I use an alternate representation known as "fixed point" Fixed-point arithmetic - Wikipedia Fixed-point takes 32 bits and can encode values from -32768.0 to +32768.0, with 15 bits for whole part and 16 bits for fractional part after decimal point, holding also sign. This is good for computation and very efficient on storage so it is used in many places in LittleCMS.

And here comes what this function cmsToFixedDomain() does: given a value in 16 bits from 0 to 0xffff, it returns the signed, 15.16 fixed-point representation of this number in 0..1.0 domain. Zero is encoded as 0.0 and 0xffff is encoded as 1.0. Remaining values are adjusted with proper rounding. No less.

At that point the bell in my head buzzed louder. How is possible if this function is supposed to handle 0..0xffff, which are positive numbers, fails with a negative number?

Looking at the parameter type, "int" makes to think the function was designed to deal with negatives. Which is actually untrue. This function uses "int" because at the time I wrote it, using native integer was the fastest way. It gives enough space to do the computation and return a 32 bits value. "int" is checked elsewhere to be of 32 bits at least.

The source of the error seems then to be not the function itself but in the caller. This function was never designed to handle negative numbers and it is certainly being called with a negative. A big one, indeed.

Let's explore the next frame of the stack dump, where we can find this gem:

```
Rest = c1 * rx + c2 * ry + c3 * rz;

Tmp1[OutChan] = (cmsUInt16Number) ( c0 +
                                    ROUND_FIXED_TO_INT(_cmsToFixedDomain(Rest)));
```

“Rest” contains the remainder from a tetrahedral interpolation; the code is meant to extract its integer part. The mistake is assuming Rest will always be in the range 0..0xFFFF while it can fall in that range, it may also be negative.

Having said this, I wonder how this issue was not evident on all the unit testing. And here comes my surprise: if you let overflow the value, it works as expected!

Obviously, this is not how it should work. C99 spec allows overflow on unsigned integers and leaves signed overflow as undefined. Undefined is scary, because one of the possible actions is to rise an FPE_INTOVF. I **never** found an implementation doing that, but it is still possible because it is described in the C99 spec.

So, was this issue a real vulnerability? Unless the compiler uses FPE_INTOVF I don’t think so. The funny part is most compilers just do the modulo thing, same as with unsigned and this works just fine. Color comes fine!

Solving the issue

The chosen solution was to align this function with the tetrahedral interpolation implementation used elsewhere (notably for 3D spaces). The replacement code is as follows:

```
Rest += 0x8001;
Tmp1[OutChan] = (cmsUInt16Number) c0 + ((Rest + (Rest>>16))>>16);
```

The new code is almost identical, differing only for two values out of 0xFFFF, and it also removes an integer division. With this fix all unit tests pass, including an assertion that previously detected negative values in cmsToFixedDomain(). It likely improves throughput for CMYK-on-input color transforms, but the exact performance gain is unknown because measuring it would require specialized MCU and small-processor hardware.

It is solved in GitHub and will make its way to incoming lcms2-2.18

Special thanks

Many thanks to the AIXCC guys for using my pet’s code as something to scrutinize. To David Korczynski for proving the details and to Derek Zimmer and the OSTIF to let me know about the issue.