Uninitialized garbage on ia64 can be deadly

devblogs.microsoft.com/oldnewthing/20040119-00

January 19, 2004

Raymond Chen

On Friday, we talked about some of the bad things that can happen if you call a function with the wrong signature. The ia64 introduces yet another possible bad consequence of a mismatched function signature which you may have thought was harmless.

The CreateThread function accepts a LPTHREAD_START_ROUTINE, which has the function signature

```
DWORD CALLBACK ThreadProc(LPVOID lpParameter);
```

One thing that people seem to like to do is to take a function that returns void and cast it to a LPTHREAD_START_ROUTINE. The theory is, "I don't care what the return value is, so I may as well use a function that doesn't have a return value. The caller will get garbage, but that's okay; garbage is fine." Here one web page that contains this mistake:

```
void MyCritSectProc(LPVOID /*nParameter*/)
{ ... }
hMyThread = CreateThread(NULL, 0,
             (LPTHREAD_START_ROUTINE) MyCritSectProc,
             NULL, 0, &MyThreadID);
```

This is hardly the only web page that supplies buggy sample code. Here's sample code from a course at Old Dominion University that makes the same mistake, and sample code from Long Island University, It's like shooting fish in a barrel. Just google for CreateThread LPTHREAD_START_ROUTINE and pretty much all the hits are people calling CreateThread incorrectly. Even sample code in MSDN gets this wrong. Here's a whitepaper that misdeclares both the return value and the input parameter in a manner that will crash on Win64,

And it's all fun until somebody gets hurt.

On the ia64, each 64-bit register is actually 65 bits. The extra bit is called "NaT" which stands for "not a thing". The bit is set when the register does not contain a valid value. Think of it as the integer version of the floating point NaN.

The NaT bit gets set most commonly from speculative execution. There is a special form of load instruction on the ia64 which attempts to load the value from memory, but if the load fails (because the memory is paged out or the address is invalid), then instead of raising a page fault, all that happens is that NaT bit gets set, and execution continues.

All mathematical operations on NaT just produce NaT again.

The load is called "speculative" because it is intended for speculative execution. For example, consider the following imaginary function:

```
void SomeClass::Sample(int *p)
{
  if (m_ready) {
    DoSomething(*p);
  }
}
```

The assembly for this function might go like this:

```
SomeClass::Sample
      alloc r35=ar.pfs, 2, 2, 1 // 2 input, 2 locals, 1 output
      mov r34, rp               // save return address
      ld4 r32=[r32]             // fetch m_ready
      ld4.s r36=[r33];;         // speculative load of *p
      cmp.eq p14, p15=r0, r32   // m_ready == 0?
(p15) chk.s r36=[r33]           // if not, validate r36
(p15) br.call rp=DoSomething    //         call
      mov rp=r34;;              // return return address
      mov.i ar.pfs=r35          // clean registers
      br.ret rp;;               // return
```

I suspect most of you haven't seen ia64 assembly before. Since this isn't an article on ia64 assembly, I'll gloss over the details that aren't relevant to my point.

After setting up the register frame and saving the return address, we load the value of m_ready and also perform a speculative load of *p into the r36 register. Notice that we are starting to execute the "true" branch of the "if" statement before we even know whether the condition is true! That's why this is known as speculative execution.

(Why do this? Because memory access is slow. It is best to issue memory accesses as far in advance of their use as possible, so you don't sit around stalled on RAM.)

We then check the value of m_ready, and if it is nonzero, we execute the two lines marked with (p15). The first is a "chk.s" instruction which says, "If the r36 register is NaT, then perform a nonspeculative load from [r33]; otherwise, do nothing."

So if the speculative load of *p had failed, the chk.s instruction will attempt to load it for real, raising the page fault and allowing the memory manager to page the memory back in (or to let the exception dispatcher raise the STATUS_ACCESS_VIOLATION).

Once the value of the r36 register has been settled once and for all, we call DoSomething. (Since we have two input registers [r32, r33] and two local registers [r34, r35], the output register is r36.)

After the call returns, we clean up and return to our own caller.

Notice that if it turns out that m_ready was FALSE, and the access of *p had failed for whatever reason, then the r36 register would have been left in a NaT state.

And that's where the danger lies.

For you see, if you have a register whose value is NaT and you so much as breathe on it the wrong way (for example, try to save its value to memory), the processor will raise a STATUS_REG_NAT_CONSUMPTION exception.

(There do exist some instructions that do not raise an exception when handed a NaT register. For example, all arithmetic operations support NaT; they just produce another NaT as the result. And there is a special "store to memory, even if it is NaT" instruction, which is handy when dealing with variadic functions.)

Okay, maybe you can see where I'm going with this. (It sure is taking me a long time.)

Suppose you're one of the people who take a function returning void and cast it to a LPTHREAD_START_ROUTINE. Suppose that function happens to leave the r8 register as NaT, because it ended with a speculative load that didn't pan out. You now return back to kernel32's thread dispatcher with NaT as the return value. Kernel32 then tries to save this value as the thread exit code and raises a STATUS_REG_NAT_CONSUMPTION exception.

Your program dies deep inside kernel and you have no idea why. Good luck debugging this one!

There's an analogous problem with passing too few parameters to a function. If you pass too few parameters to a function, the extra parameters might be NaT. And the great thing is, even if the function is careful not to access that parameter until some other conditions are met, the compiler may find that it needs to spill the parameter, thereby raising the STATUS_REG_NAT_CONSUMPTION exception.

I've actually seen it happen. Trust me, you don't want to get tagged to debug it.

The ia64 is a very demanding architecture. In tomorrow's entry, I'll talk about some other ways the ia64 will make you pay the penalty when you take shortcuts in your code and manage to skate by on the comparatively error-forgiving i386.

Raymond Chen

Follow