The memcmp function reports the result of the comparison at the point of the first difference, but it can still read past that point

devblogs.microsoft.com/oldnewthing/20101014-00

October 14, 2010

Raymond Chen

This story originally involved a more complex data structure, but that would have required too much explaining (with relatively little benefit since the data structure was not related to the moral of the story), so I'm going to retell it with double null-terminated strings as the data structure instead.

Consider the following code to compare two double-null-terminated strings for equality:

```
size_t SizeOfDoubleNullTerminatedString(const char *s)
{
  const char *start = s;
  for (; *s; s += strlen(s) + 1) { }
  return s - start + 1;
}
BOOL AreDoubleNullTerminatedStringsEqual(
    const char *s, const char *t)
{
 size_t slen = SizeOfDoubleNullTerminatedString(s);
 size_t tlen = SizeOfDoubleNullTerminatedString(t);
 return slen == tlen && memcmp(s, t, slen) == 0;
}
```

"Aha, this code is inefficient. Since the `memcmp` function stops comparing as soon as it finds a difference, I can skip the call to `SizeOfDoubleNullTerminatedString(t)` and simply write

```
BOOL AreDoubleNullTerminatedStringsEqual(
    const char *s, const char *t)
{
 return memcmp(s, t, SizeOfDoubleNullTerminatedString(s)) == 0;
}
```

because we can never read past the end of `t`: If the strings are equal, then `tlen` will be equal to `slen` anyway, so the buffer size is correct. And if the strings are different, the difference will be found at or before the end of `t`, since it is not possible for a double-null-terminated string to be a prefix of another double-null-terminated string. In both cases, we never read past the end of `t`."

This analysis is based on a flawed assumption, namely, that `memcmp` compares byte-by-byte and does not look at bytes beyond the first point of difference. The `memcmp` function makes no such guarantee. It is permitted to read all the bytes from both buffers before reporting the result of the comparison.

In fact, most implementations of `memcmp` *do* read past the point of first difference. Your typical library will try to compare the two buffers in register-sized chunks rather than byte-by-byte. (This is particularly convenient on x86 thanks to the block comparison instruction `rep cmpsd` which compares two memory blocks in `DWORD`-sized chunks, and x64 doubles your fun with `rep cmpsq`.) Once it finds two chunks which differ, it then studies the bytes within the chunks to determine what the return value should be.

(Indeed, people with free time on their hands or simply enjoy a challenge will try to outdo the runtime library with fancy-pants `memcmp` algorithms which compare the buffers in larger-than-normal chunks by doing things like comparing via SIMD registers.)

To illustrate, consider an implementation of `memcmp` which uses 4-byte chunks. Typically, memory comparison functions do some preliminary work to get the buffers aligned, but let's ignore that part since it isn't interesting. The inner loop goes like this:

```
while (length >= 4)
{
 int32 schunk = *(int32*)s;
 int32 tchunk = *(int32*)t;
 if (schunk != tchunk) {
   -- difference found - calculate and return result
 }
 length -= 4;
 s += 4;
 t += 4;
}
```

Let's compare the strings `s = "a\0b\0\0"` and `t = "a\0\0"`. The size of the double-null-terminated string `s` is 4, so the memory comparison goes like this: First we read four bytes from `s` into `schunk`, resulting in (on a little-endian machine) `0x00620061`. Next, we read four bytes from `t` into `tchunk`, resulting in `0x??000061`. Oops, we read one byte past the end of the buffer.

If `t` happened to sit right at the end of a page, and the next page was uncommitted memory, then you take an access violation while trying to read `tchunk`. Your optimization turned into a crash.

Remember, when you say that a buffer is a particular size, the basic ground rules of programming say that it really has to be that size.

Raymond Chen

Follow