The case of the string being copied from a mysterious pointer to invalid memory, revisited

devblogs.microsoft.com/oldnewthing/20240911-00

September 11, 2024

Some time ago, I wrote about the case of the string being copied from a mysterious pointer to invalid memory. The primary purpose of that article was to show how to use Application Verifier to investigate a memory corruption bug caused by a race condition. Another way to investigate this is using a tool like Address Sanitizer, but Application Verifier has the advantage of not requiring that the code be recompiled, so you can ask a customer to turn it on at their site to investigate a problem that you can't reproduce in-house, and it can verify code you can't recompile, like an external library.

Some people had issues with my proposed solution for ensuring thread-safe access to the member variable, which is to acquire the lock only around the reads and writes to the member, and to drop the lock when calling the `SlowGetId()` function.

```
std::string GetId()
{
    if (auto lock = std::shared_lock(m_sharedMutex);
        !m_uniqueId.empty())
    {
        return m_uniqueId;
    }

    auto uniqueId = SlowGetId();
    auto lock = std::unique_lock(m_sharedMutex);
    if (m_uniqueId.empty()) {
        m_uniqueId = std::move(uniqueId);
    }
    return m_uniqueId;
}
```

An alternative design would be to hold the lock across the entire operation:

```
std::string GetId()
{
    auto lock = std::unique_lock(m_sharedMutex);
    if (!m_uniqueId.empty())
    {
        return m_uniqueId;
    }

    auto uniqueId = SlowGetId();
    if (m_uniqueId.empty()) {
        m_uniqueId = std::move(uniqueId);
    }
    return m_uniqueId;
}
```

And you could further simplify this by using a `std::once_flag` for code that should run exactly once:

```
std::string GetId()
{
    std::call_once(m_once, [this] {
        m_uniqueId = SlowGetId();
    });
    return m_uniqueId;
}
```

I decided against these options because I didn't know how safe it was to hold the lock across the call to `SlowGetId()`. For example, if `SlowGetId()` opens a re-entrancy window, then the re-entrant call will deadlock against itself. Since I wasn't familiar with the rules for this code, I played it safe and preserved any existing re-entrant behavior. If `SlowGetId()` calls a function that in turn conditionally calls `GetId()`, the old code would just call `SlowGetId()` again, and the second call might not call `GetId()`, thus breaking the mutual recursion. As the stack unwinds, the `m_uniqueId` gets set twice, but at least it gets set successfully. Another case where you might see this re-entrancy is if somebody inside `SlowGetId()` makes a cross-thread COM call and has to pump messages while waiting for the answer, and during that time, an inbound call from another thread wants to get the ID. In both of these cases, holding the lock across `SlowGetId()` would introduce a hang. If the concurrent calls are rare but the reentrant calls are common, then the cure ended up being worse than the disease.

It has also been noted that if the method had been marked `const`, then the modification of `m_uniqueId` would have been disallowed. In the original case (which I simplified for expository purposes), the `GetId()` was a COM method, and COM methods are never `const`. The concept of `const` methods doesn't exist in the COM ABI. An implementation is welcome to make a method behave as if it were `const`, but COM does not *require* any method to be `const`. Whether the method call changes state is an implementation detail.