The case of the crash when destructing a std::map

devblogs.microsoft.com/oldnewthing/20240927-00

September 27, 2024

A customer reported that they were getting crashes while destructing a `std::map`.

Here’s the point of the crash:

```
eax=245bd25c ebx=00004d33 ecx=31feece4 edx=00c40000 esi=00000000 edi=31feece4
eip=563397db esp=245bd250 ebp=245bd268 iopl=0         nv up ei ng nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010286
contoso!std::_Tree<⟦...⟧>::~_Tree+0x2b:
563397db cmp     byte ptr [esi+0Dh],0                       ds:002b:0000000d=??

0:054> k9
ChildEBP RetAddr
(Inline) --------     contoso!std::_Tree_val<⟦...⟧>::_Erase_tree [xtree @ 1073]
(Inline) --------     contoso!std::_Tree_val<⟦...⟧>::_Erase_head+0x5 [xtree @ 1073]
245bd268 56338313     contoso!std::_Tree<⟦...⟧>::~_Tree+0x2b [xtree @ 1073]
245bd290 56362695     contoso!Contoso::IdCollection::~IdCollection+0x163
245bd2b4 564c8510     contoso!Contoso::LogEntry::~LogEntry+0xd5
245bd2e0 5691a999     contoso!Contoso::LogEntries::Cleanup+0x90
(Inline) --------     contoso!Contoso::Log::Flush+0x8
245be134 56916406     contoso!Contoso::TaskRunner::RunTasks+0x3949
```

Okay, so we are trying to destruct a `_Tree`, which is the internal class that acts as the basis for `map`, `multimap`, `set`, and `multiset`. In this case, we are destructing a `std::map`.

I wasted invested a good amount of time reading the STL source code in order to figure out the internal structure of `std::map`. You can read the linked article for the details, but the short version is that the map consists of a sentinel node where

- sentinel.left = first node in the tree
- sentinel.right = last node in the tree
- sentinel.parent = root node of the tree

For the root node and other nodes of the tree, the left, right, and parent have their normal meanings.

If there is no root node, left subtree, or right subtree, then the corresponding member contains a pointer to the sentinel node. (The use of a sentinel node is a standard computer science trick which removes the need to add null pointer checks everywhere.)

We can read the STL code to see how tree destruction occurs.

We start with the `_Tree` destructor:

```
~_Tree() noexcept {
    const auto _Scary = _Get_scary();
    _Scary->_Erase_head(_Getal());
}
```

The `Scary` stuff is just to scare you. It's just getting a value from the tree.

```
_Scary_val* _Get_scary() noexcept {
    return _STD addressof(_Mypair._Myval2._Myval2);
}
```

Meanwhile, `_Erase_head` goes like this:

```
template <class _Alnode>
void _Erase_head(_Alnode& _Al) noexcept {
    this->_Orphan_all();
    _Erase_tree(_Al, _Myhead->_Parent);
    _Alnode::value_type::_Freenode0(_Al, _Myhead);
}
```

This just erases the tree and then frees the sentinel node. So the real excitement is in _Erase_tree:

```
template <class _Alnode>
void _Erase_tree(_Alnode& _Al, _Nodeptr _Rootnode) noexcept {
    while (!_Rootnode->_Isnil) { // free subtrees, then node
        _Erase_tree(_Al, _Rootnode->_Right);
        _Alnode::value_type::_Freenode(_Al,
            _STD exchange(_Rootnode, _Rootnode->_Left));
    }
}
```

Erasing the tree consists of recursively erasing the right node, freeing the node, and tail recursing on the left node.

Now, the crashing instruction is cmp byte ptr [esi+0Dh],0, which is obviously the `_Rootnode->_Isnil`. "Obviously" because it is the only place we check a byte. All other operations are on pointers.

But let's look at the crash in the context of the whole function just to make sure, and see what else we can learn.

```
// Function prologue nonsense

contoso!std::_Tree<⟦...⟧>::~_Tree
563397b0 push    ebp
563397b1 mov     ebp,esp
563397b3 push    0FFFFFFFFh
563397b5 push    offset contoso!___guard_check_icall_thunk+0x25d0 (56d0e690)
563397ba mov     eax,dword ptr fs:[00000000h]
563397c0 push    eax
563397c1 push    esi
563397c2 push    edi
563397c3 mov     eax,dword ptr [contoso!__security_cookie (5427c040)]
563397c8 xor     eax,ebp
563397ca push    eax
563397cb lea     eax,[ebp-0Ch]
563397ce mov     dword ptr fs:[00000000h],eax

// inlined _Erase_head

563397d4 mov     edi,ecx                ; edi = this
563397d6 mov     esi,dword ptr [edi]    ; esi = _Myhead
563397d8 mov     esi,dword ptr [esi+4]  ; esi = _Myhead.Parent

// inlined _Erase_tree (esi = _Rootnode)

563397db cmp     byte ptr [esi+0Dh],0   ; while (!_Rootnode->_Isnil) ← CRASHED HERE
563397df jne     contoso!std::_Tree<⟦...⟧>::~_Tree+0x51 (56339801) ; break out of
loop

loop:
// Recursive call to _Erase_tree
563397e1 push    dword ptr [esi+8]      ; _Rootnode->_Right
563397e4 mov     ecx,edi                ; outbound this = inbound this
563397e6 push    edi                    ; allocator
563397e7 call    contoso!std::_Tree_val<⟦...⟧>::_Erase_tree (5633b450) ; erase the
subtree

563397ec mov     eax,esi                ; delete the old _Rootnode
563397ee mov     esi,dword ptr [esi]    ; fetch _Rootnode->_Left for tail recursion
563397f0 push    18h
563397f2 push    eax
563397f3 call    contoso!operator delete (56d08334) ; free the old _Rootnode
563397f8 add     esp,8

563397fb cmp     byte ptr [esi+0Dh],0   ; while (!_Rootnode->_Isnil)
563397ff je      contoso!std::_Tree<⟦...⟧>::~_Tree+0x31 (563397e1)

// end of _Erase_head, now free the sentinel node

56339801 push    18h
56339803 push    dword ptr [edi]
56339805 call    contoso!operator delete (56d08334)
```

```
5633980a add     esp,8

5633980d mov     ecx,dword ptr [ebp-0Ch]
56339810 mov     dword ptr fs:[0],ecx
56339817 pop     ecx
56339818 pop     edi
56339819 pop     esi
5633981a mov     esp,ebp
5633981c pop     ebp
5633981d ret
```

Okay, so we were right that the crash was on the test of `_Rootnode->_Isnil`, but we also learned that this is the test that occurs before entering the loop body for the first time. (The tests that occur on subsequent iterations come later in the function.)

This is great, because it tells us that no changes to the tree have been made yet. We aren't looking at a tree in a temporarily invalid state because the destructor is messing with it. Instead, the tree is still its originally-corrupted state.

The crash is on a null `_Rootnode`, and that came from `_Myhead._Parent`, so our tree must have a null `_Myhead._Parent`, which is not allowed. (An empty tree has a `_Myhead._Parent` that points back to the sentinel node.)

Let's see what we have in the tree.

```
0:054> ?? this->_Mypair._Myval2._Myval2
class std::_Tree_val<⟦...⟧>
   +0x000 _Myhead          : 0x1ca4f280 std::_Tree_node<⟦...⟧>
   +0x004 _Mysize          : 0
```

Okay, so this tree is empty (_Mysize is zero).

```
0:054> ?? this->_Mypair._Myval2._Myval2._Myhead
struct std::_Tree_node<⟦...⟧> * 0x1ca4f280
   +0x000 _Left            : 0xc00000b0 std::_Tree_node<⟦...⟧>
   +0x004 _Parent          : (null)
   +0x008 _Right           : 0x1ca4f280 std::_Tree_node<⟦...⟧>
   +0x00c _Color           : 1 ''
   +0x00d _Isnil           : 1 ''
   +0x010 _Myval           : std::pair<int const, enum ChannelType>
```

As expected, this is the sentinel node, (_Isnil is 1). What's not expected is that the `_Parent` is null, and the `_Left` is corrupted. The `_Right` is okay: It points back to the sentinel node.

That corrupted value for `_Left` looks really suspicious: It is of the form `0xc000nnnn`, which is the range used by `NTSTATUS` codes. And if we dump the node as bytes, we can see that the corruption is restricted to just those first two dwords.

```
0:054> dc 1ca4f280 L10
1ca4f280  c00000b0 00000000 1ca4f280 00000101  ................
          ^^^^^^^^ ^^^^^^^^ ^^^^^^^^ ^^^^^^^^
         corrupted corrupted    okay     okay
```

What is the `NTSTATUS` code that got written?

```
C:\>certutil /error 0xc00000b0
0xc00000b0 (NT: 0xc00000b0 STATUS_PIPE_DISCONNECTED) -- 3221225648 (-1073741648)
Error message text: The specified named pipe is in the disconnected state.
CertUtil: -error command completed successfully.
```

To me, this looks like what happens when an overlapped I/O completes. The first two fields of the `OVERLAPPED` structure are updated by the kernel at the completion of the I/O, and the two things it writes are the status code and the number of bytes transferred (which is unsurprisingly zero seeing as an error occurred).

My theory was that this program at some point issued an overlapped I/O and freed the `OVERLAPPED` structure associated with the I/O before the I/O completed. That memory then got reused to hold the `std::map` sentinel node, and then the I/O completed, and the kernel wrote the I/O result into what it thought was the `OVERLAPPED` structure (but is now the `std::map` sentinel node), thereby corrupting the sentinel node.

The customer said, “We don’t use overlapped I/O, but maybe one of the libraries we use does.”

They provided their source code in the form of a massive 5 gigabyte ZIP file. Thankfully, they also gave me access to their online repo, so I could use the search functionality in their repo hosting provider.

I searched their code for `OVERLAPPED` and found a few references. A lot of them were just the word “overlapped” being used in a comment, but it wasn’t long before I found an actual `OVERLAPPED` structure, and here it is.

```
void Channel::ReadData(⟦...⟧)
{
    ⟦...⟧

    OVERLAPPED o{};
    o.hEvent = m_readCompleteEvent;

    if (ReadFile(m_file, m_buffer, m_bufferSize, &actual, &o)) {
        // completed synchronously
        ⟦...⟧
    } else if (GetLastError() != ERROR_IO_PENDING) {
        ⟦ handle various error conditions ⟧
    } else {
        // Wait for I/O to complete.
        switch (WaitForSingleObject(o.hEvent, IO_TIMEOUT)) {
        case WAIT_OBJECT_0:
            ⟦... process the results ...⟧
            break;

        case WAIT_ABANDONED:
            ⟦... deal with the error ...⟧
            break;

        case WAIT_TIMEOUT:
            break;

        default:
            ⟦... unexpected error ...⟧
            break;
        }
    }
}
```

After they issue the overlapped read, they wait up to `IO_TIMEOUT` (1000) milliseconds for an answer. If there is no answer after that time, they just give up and return.[1]

Do you see the problem?

They never cancel the I/O and wait for it complete. They just abandon the I/O and return immediately.

When the function returns, the `OVERLAPPED` structure on the stack becomes available for reuse, and then when the I/O finally does complete, the kernel writes the I/O status to memory that has since been repurposed for something else. (It also writes the data to the original `m_buffer` which might also have been freed by the time the I/O completes.)

I'm not sure what they were thinking here. They started an I/O and just walked away. How does the kernel know that it should stop executing the I/O and stop writing the I/O results back into application memory?

It’s like booking a demolition company to knock down your house, and they say, “We’re really busy right now, but we’ve added you to our schedule. We can’t promise an exact date, but trust us, we’ll show up to knock down your house when it’s your turn.” You get tired of waiting for them and just sell the house and move somewhere else. Eventually, that demolition company will show up and knock down that house, even though it now belongs to somebody else.

When I discussed this bug investigation with some colleagues, one of them remarked, “Wow, how lucky you were! The very first hit was the memory corruption bug you were looking for.”

I replied, “As it turns out, it wasn’t luck.” This code base was a target-rich environment. Every single overlapped I/O had this same bug: Nobody ever cancelled I/O before abandoning it! If the I/O didn’t complete within a specified timeout, the code always simply walked away from it.

(Note that this code is really lucky that the I/O eventually failed. If it had succeeded, they would also have corrupted whatever object was placed in the memory formerly used as the I/O output buffer!)

But wait, this is stack corruption. The original problem was heap corruption. Even though this is bad, it’s not the bug that caused the crash.

I found two places that performed asynchronous I/O into an `OVERLAPPED` structure on the heap. Here’s one of them:

```
class Writer
{
    〚...〛

    OVERLAPPED m_overlapped;

    〚...〛
};

ErrorCode Writer::Write(void* buffer, unsigned size)
{
    〚...〛

    if (!WriteFile(m_target, buffer, size,
                   &actual, &m_overlapped)) {
        return ErrorCode::WriteFailed;
    }

    if (GetLastError() != ERROR_IO_PENDING) {
        return ErrorCode::WriteFailed;
    }

    if (WaitForSingleObject(o.hEvent, 5000)
            == WAIT_TIMEOUT) {
        return ErrorCode::WriteTimeout;
    }
}
```

This issues an overlapped write to the `m_target` and waits 5 seconds for the write to complete. if it doesn't complete, then it just abandons the operation and returns a failure code.

What's happening is that if this write operation takes more than five seconds, the failure code propagates up the call stack, and I guess it destructs the `Writer`, allowing the memory for `m_overlapped` to be reused by the `IdCollection`, which then gets corrupted when the I/O finally completes.

Notice that the crash is in the logging code, and the log entry is probably created immediately after the `Writer` is freed, so it ends up reusing that memory. And then the overlapped I/O completes and updates what it thought was an `OVERLAPPED` structure but which is now the map sentinel node.

The fix is to make sure that when we decide to abandon an I/O operation, we cancel it and wait for the I/O to complete. (It will probably complete with `ERROR_CANCELLED`.)

For example, we could do this:

```
ErrorCode Writer::Write(void* buffer, unsigned size)
{
    ⟦...⟧

    if (!WriteFile(m_target, buffer, size,
                &actual, &m_overlapped)) {
        return ErrorCode::WriteFailed;
    }

    if (GetLastError() != ERROR_IO_PENDING) {
        return ErrorCode::WriteFailed;
    }

    if (WaitForSingleObject(o.hEvent, 5000)
            == WAIT_TIMEOUT) {
        CancelIoEx(m_target, &m_overlapped);
        GetOverlappedResult(m_target, &m_overlapped,
            &actual, TRUE);
        return ErrorCode::WriteTimeout;
    }
}
```

[1] Yes, this code tests for `WAIT_ABANDONED`, even though that error code will never be returned when waiting on event. The `WAIT_ABANDONED` error code is used only by mutexes.