The case of the crash when trying to erase an element from a std::set

devblogs.microsoft.com/oldnewthing/20250117-00

January 17, 2025

Today, we'll look at a crash that occurred when trying to erase an element from a `std::set`.

```
rax=000001f565bc046e rbx=000001f589b20340 rcx=000001f565bc046e
rdx=000000e6658feca8 rsi=000001f589b20690 rdi=000001f589b203c0
rip=00007ffdd4726bc4 rsp=000000e6658fec30 rbp=0000388a1713ab55
 r8=000001f589b895d0  r9=000001f589b895d0 r10=000001f589000140
r11=0000000000000000 r12=0000000000000001 r13=000000007ffe0385
r14=0000000000000000 r15=000001f589b8f900

LitWare!std::_Tree<std::_Tset_traits<WidgetWatcher *,
        std::less<WidgetWatcher *>,
        std::allocator<WidgetWatcher *>,0> >::_Eqrange+0x14
    [inlined in LitWare!std::_Tree<std::_Tset_traits<
        WidgetWatcher *,std::less<WidgetWatcher *>,
        std::allocator<WidgetWatcher *>,0> >::erase+0x18]:
00007ffd`d4726bc4 cmp     byte ptr [rax+19h],r11b ds:000001f5`65bc0487=??
```

The stack trace has some information about how we got here.

```
LitWare!std::_Tree<std::_Tset_traits<Widget *,
        std::less<Widget *>,
        std::allocator<Widget *>,0> >::_Eqrange+0x14
LitWare!std::_Tree<std::_Tset_traits<Widget *,
        std::less<Widget *>,
        std::allocator<Widget *>,0> >::erase+0x18
LitWare!Widget::~Widget+0xc8
LitWare!Widget::`scalar deleting destructor'+0x14
LitWare!DestroyWidget+0x15
Fabrikam!Doodad::~Doodad+0x75
Fabrikam!Doodad::`scalar deleting destructor'+0x14
Fabrikam!Doodad::Release+0x40
Contoso!Gadget::~Gadget+0x66
ucrtbase!<lambda_⟦...⟧>::operator()+0xa5
ucrtbase!__crt_seh_guarded_call<int>::operator()<⟦...⟧>+0x3b
ucrtbase!__acrt_lock_and_call+0x1c
ucrtbase!_execute_onexit_table+0x3d
Contoso!dllmain_crt_process_detach+0x45
Contoso!dllmain_dispatch+0xe6
ntdll!LdrpCallInitRoutine+0xb0
ntdll!LdrShutdownProcess+0x260
ntdll!RtlExitUserProcess+0x114
kernel32!FatalExit+0xb
ucrtbased!exit_or_terminate_process+0x3a
ucrtbased!common_exit+0x85
ucrtbased!exit+0x16
```

The top of the stack tells us that we are trying to erase an element from a `std::set`. This happened in the `Widget` destructor:

```
struct Widget
{
    ⟦ ... ⟧

    // For debugging purposes, keep track of all of the Widgets.
    static wil::srwlock s_lock;
    static std::set<Widget*> s_allWidgets;
};

Widget::~Widget()
{
    auto guard = s_lock.lock_exclusive();
    s_allWidgets.erase(this);
}
```

The idea is that whenever we create a `Widget`, we store its address in the `s_allWidgets` set, and when we destroy one, we remove it from the set. The comment notes that this is just for debugging purposes, so that when we want to see what all the Widgets are doing, we can walk through the set to find each one.

Okay, so now that we have a general idea of what this code is trying to do, let's go back and study the crash.

First, what is the Widget being destructed?

```
0:000> .frame 2
02 LitWare!Widget::~Widget+0xc8
0:000> dv
           this = 0x000001f5`89b20340
```

Okay, remember that number.

Next, is the `std::set` corrupted?

```
0:000> dx LitWare!Widget::s_allWidgets
LitWare!Widget::s_allWidgets : { size=0x1 }
    [<Raw View>]
    [comparator]     : less
    [allocator]      : allocator
0:000> ?? LitWare!Widget::s_allWidgets
class std::set<Widget *,std::less<Widget *>,std::allocator<Widget *> >
   +0x000 _Mypair          : std::_Compressed_pair<⟦...⟧>
0:000> ?? LitWare!Widget::s_allWidgets._Mypair
class std::_Compressed_pair<⟦...⟧>
   +0x000 _Myval2          : std::_Compressed_pair<⟦...⟧>
0:000> ?? LitWare!Widget::s_allWidgets._Mypair._Myval2
class std::_Compressed_pair<⟦...⟧>
   +0x000 _Myval2          : std::_Compressed_pair<⟦...⟧>
0:000> ?? LitWare!Widget::s_allWidgets._Mypair._Myval2._Myval2
class std::_Tree_val<std::_Tree_simple_types<Widget *> >
   +0x000 _Myhead          : 0x000001f5`89b895d0 std::_Tree_node<Widget *,void *>
   +0x008 _Mysize          : 1
```

Okay, after digging through a bunch of compressed pairs, we finally get to the goods. There is one element in the set, and we can look at the sentinel node.

```
0:000> ?? LitWare!Widget::s_allWidgets._Mypair._Myval2._Myval2._Myhead
struct std::_Tree_node<Widget *,void *> * 0x000001f5`89b895d0
   +0x000 _Left            : 0x000001f5`89b800cb std::_Tree_node<Widget *,void *>
   +0x008 _Parent          : 0x000001f5`65bc046e std::_Tree_node<Widget *,void *>
   +0x010 _Right           : 0x000001f5`89b846e0 std::_Tree_node<Widget *,void *>
   +0x018 _Color           : 1 ''
   +0x019 _Isnil           : 1 ''
   +0x020 _Myval           : (null)
```

This already looks bad, because the `_Left` and `_Parent` pointers are misaligned. Furthermore, if the set has only one element, then the root (`_Parent`), left, and right nodes should all be the same, but all of these pointers are different.

The not-yet-obviously-corrupted pointer is `_Right`, and chasing through shows a similar type of corruption:

```
0:000> ?? LitWare!Widget::s_allWidgets._Mypair._Myval2._Myval2._Myhead->_Right
struct std::_Tree_node<Widget *,void *> * 0x000001f5`89b846e0
   +0x000 _Left            : 0x000001f5`89b80268 std::_Tree_node<Widget *,void *>
   +0x008 _Parent          : 0x000001f5`f69fb889 std::_Tree_node<Widget *,void *>
   +0x010 _Right           : 0x000001f5`89b895d0 std::_Tree_node<Widget *,void *>
   +0x018 _Color           : 1 ''
   +0x019 _Isnil           : 0 ''
   +0x020 _Myval           : 0x000001f5`89b20340 Widget *
0:000>
```

The first two pointers are corrupted, but the rest looks okay. (Notice that the `_Right` points back to the sentinel node, as expected, and the `_Myval` is the pointer to the `Widget` we are trying to remove.)

When I see a memory block that has had two pointers unexpectedly written to the start, my instincts tell me that I might be looking at a freed memory block. It is a common design in heap managers to store metadata about a freed memory block in the freed memory block itself. Often, the free blocks are kept in a doubly-linked list, which explains why the corruption is seen as two pointers.

Okay, so my working theory is that this is freed memory, which implies that the `std::set` has already destructed. This theory is supported by the stack trace:

```
LitWare!std::_Tree<std::_Tset_traits<Widget *,
        std::less<Widget *>,
        std::allocator<Widget *>,0> >::_Eqrange+0x14
LitWare!std::_Tree<std::_Tset_traits<Widget *,
        std::less<Widget *>,
        std::allocator<Widget *>,0> >::erase+0x18
LitWare!Widget::~Widget+0xc8
LitWare!Widget::`scalar deleting destructor'+0x14
LitWare!DestroyWidget+0x15
Fabrikam!Doodad::~Doodad+0x75
Fabrikam!Doodad::`scalar deleting destructor'+0x14
Fabrikam!Doodad::Release+0x40
Contoso!Gadget::~Gadget+0x66
ucrtbase!<lambda_⟦...⟧>::operator()+0xa5
ucrtbase!__crt_seh_guarded_call<int>::operator()<⟦...⟧>+0x3b
ucrtbase!__acrt_lock_and_call+0x1c
ucrtbase!_execute_onexit_table+0x3d
Contoso!dllmain_crt_process_detach+0x45
Contoso!dllmain_dispatch+0xe6
ntdll!LdrpCallInitRoutine+0xb0
ntdll!LdrShutdownProcess+0x260
ntdll!RtlExitUserProcess+0x114
kernel32!FatalExit+0xb
ucrtbased!exit_or_terminate_process+0x3a
ucrtbased!common_exit+0x85
ucrtbased!exit+0x16
```

The process is exiting, and we are notifying `Contoso.dll`. This prompts the DLL to destruct its global variables, one of which is apparently a `Gadget`, or at least it's a global variable whose destruction leads to the destruction of a `Gadget`. The destruction of the `Gadget` in turn release a `Doodad`, which causes it to destruct, and the destructor of the `Doodad` calls `DestroyWidget` which causes the `Widget` to destruct, and that's when we notice that the `s_allWidgets` has already been destructed.

We are a victim of the static initialization order fiasco, but at the other end of the story.

People usually worry about the static initialization order fiasco at the initialization side: Making sure that objects you depend on during initialization have themselves been initialized. But what goes up must come down, and you also have to make sure that the object you depend on during destruction have themselves not yet been destructed.

In this case, the `s_allWidgets` was destructed as part of `LitWare.dll`'s cleanup, even though `Contoso.dll` still had plans on using it.

This can happen even in the absence of DLLs at all. Objects with static storage duration are destructed in reverse order of construction, so we may have this:

```
// File 1 - global variable, suppose this constructs first
struct Gadget
{
    std::unique_ptr m_widget;
};
Gadget mainGadget;

// File 2 - global variable, suppose this constructs second
std::set<Widget*> s_allWidgets;
```

At program startup, we construct the `mainGadget`. The `Gadget` constructor doesn't need a `Widget` right away, so there's no static initialization order fiasco. Later, we decide that the main gadget needs a widget, so we do a `m_widget = std::make_unique<Widget>()`, and this returns the newly-constructed `Widget`, as well as registering the widget in the `s_allWidgets` set.

At destruction, the `s_allWidgets` destructs first, and then when the main gadget destructs, it destructs the now-nonempty `m_widget`, which destructs the `Widget` and tries to unregister it from the already-destructed `std::set`.

Using function-local statics to create the `std::set` on demand at first use avoids the static initialization order fiasco, but it doesn't help the static *destruction* order fiasco. The `std::set` was constructed *after* the `Gadget`, so it destructs *first*.

This particular component used the Windows Implementation Library, so it can use `wil::ProcessShutdownInProgress()` to skip the `Widget`-tracking code during shutdown.

```
Widget::~Widget()
{
    if (!wil::ProcessShutdownInProgress()) {
        auto guard = s_lock.lock_exclusive();
        s_allWidgets.erase(this);
    }
}
```

If you don't use WIL, you can get a similar effect by creating your own shutdown canary.

```
bool s_shuttingDown = false;
std::set<Widget*> s_allWidgets;

// Put the canary last so that it destructs first.
struct shutdown_canary
{
    ~shutdown_canary() { s_shuttingDown = true; }
} s_widgetShutdownCanary;

Widget::~Widget()
{
    if (!s_shuttingDown)
        auto guard = s_lock.lock_exclusive();
        s_allWidgets.erase(this);
    }
}
```

Bonus chatter: Afterward, I was able to confirm my theory that the memory for the `std::set` had indeed been freed.

```
0:000> !heap -p -a 0x000001f5`89b895d0
    address 000001f589b895d0 found in
    _HEAP @ 1f589000000
              HEAP_ENTRY Size Prev Flags            UserPtr UserSize - state
        000001f589b895d0 0048 0000  [00]   000001f589b895d0    00048 - (free)

0:000> !heap -p -a 0x000001f5`89b846e0
    address 000001f589b846e0 found in
    _HEAP @ 1f589000000
              HEAP_ENTRY Size Prev Flags            UserPtr UserSize - state
        000001f589b846e0 0048 0000  [00]   000001f589b846e0    00048 - (free)
```