The case of the invalid handle despite being managed by an RAII type

devblogs.microsoft.com/oldnewthing/20230615-00

June 15, 2023

Raymond Chen

There were sporadic reports of an invalid handle in a component. This component was using a “widget tracker”. The widget tracker is rather complicated, but here’s a simplified version:

```
typedef struct Tracker* TrackerHandle;

enum CallbackReason
{
    WidgetFound,
    InitialScanComplete,
    TrackerDestroyed,
}:

typedef void (CALLBACK *TrackerCallback)(int reason, int id, void* context);

struct TrackerOptions { /* not important */ };

HRESULT WINAPI CreateTracker(
    TrackerOptions const* options,
    TrackerCallback callback,
    void* context,
    TrackerHandle* trackerHandle);

void WINAPI DestroyTracker(
    TrackerHandler trackerHandle);
```

The idea is that you create a tracker, and the tracker will issue a `WidgetFound` callback for each widget it finds. Next, it issues a `InitialScanComplete` callback to let you know that the initial widget search is complete, and it’s now monitoring for any new widgets that get created, issuing `WidgetFound` for any new widgets that get discovered. Finally, when you destroy the tracker, you get a `Destroyed` callback to let you know that the tracker is destroyed and will not generate any new callbacks.

The customer created different kinds of trackers, but used a common class to help find the first widget.

```
struct FirstWidgetResults
{
  int GetWidgetId() { return widgetId; }
  void WaitForResults() { readyEvent.wait(); }
  void WaitForDestroyed() { destroyedEvent.wait(); }

  static void CALLBACK OnCallback(
      int reason, int id, void* context)
  {
    auto self = (MyTrackerResults*)context;

    switch (reason) {
    case WidgetFound:
      if (self->widgetId == INVALID_WIDGET_ID) {
        self->widgetId = id;
        self->readyEvent.Set();
      }
      break;

    case InitialScanComplete:
      self->readyEvent.Set();
      break;

    case TrackerDestroyed:
      self->destroyedEvent.Set();
      break;
    }
  }

private:
  int widgetId = INVALID_WIDGET_ID;

  wil::unique_event readyEvent{ wil::EventOptions::None };
  wil::unique_event destroyedEvent{ wil::EventOptions::None };

};
```

The idea behind this `FirstWidgetResults` class is that you use it to deal with the widget tracker callbacks:

- When the first widget is found, it remembers the ID of that widget and signals that the results are ready.
- When the initial search is complete, it signals that the results are ready. This is important if no widgets were found during the initial search, so that we don't wait forever for widgets to be found when none exist.
- When the tracker is destroyed, it signals that the helper class can safely be destructed.

They found that sometimes they got an invalid handle failure at the `destroyedEvent.Set()`. How can that be? The event handle is kept in an RAII type, so we shouldn't be having handle lifetime issues, yet the crash dumps tell us otherwise.

The call stack for the invalid handle exception looks like this:

```
ntdll!KiRaiseUserExceptionDispatcher+0x3a
KERNELBASE!SetEvent+0xd
Contoso!wil::details::SetEvent+0xa
Contoso!wil::event_t::SetEvent+0xa
Contoso!FirstWidgetResults::OnCallback+0x73
litware!WidgetTracker::RaiseCallback+0xe2
...
```

We can ask the debugger for the identity of the `FirstWidgetResult` that failed.

```
0:008> .frame 2
02 Contoso!wil::details::SetEvent+0xa
0:008> dv

0:008> .frame 3
03 Contoso!wil::event_t::SetEvent+0xa
0:008> dv

0:008> .frame 4
04 Contoso!FirstWidgetResults::OnCallback+0x73
0:008> dv
         reason = <value unavailable>
             id = <value unavailable>
        context = <value unavailable>

0:008> .frame 5
05 litware!WidgetTracker::RaiseCallback+0xe2
0:008> dv
               this = 0x000001d6`b72bbde0
              event = 0x000001d6`b72bbe58

0:008> ?? this
class WidgetTracker * 0x000001d6`b72bbde0
   +0x008 m_SRWLock        : _RTL_SRWLOCK
   +0x000 __VFN_table : 0x00007ffd`0d1c8c50
   +0x010 m_pClientContext : 0x000000a9`4e5ff298 Void
   +0x018 m_pClientCallback : 0x00007ffc`d3e35980
Contoso!FirstWidgetResults::OnCallback+0
   ...
```

The compiler optimized out the event handle and the `this` pointer for the `FirstWidget-Result`, so we had to go digging further and further back into the stack until we could recover it. We eventually were able to find it from the widget tracker itself.

```
0:008> dt -r1 contoso!FirstWidgetResults 0x000000a9`4e5ff298
   +0000 widgetId          : -739262712
   +0x008 readyEvent       : wil::unique_event
      +0x000 m_ptr            : 0x00007ffc`eeb48b58 Void
   +0x010 destroyedEvent   : wil::unique_event
      +0x000 m_ptr            : 0x00007ffc`eeaf0000 Void
```

Yeah, all of those values look like garbage. Widget IDs and kernel event handles are normally small integers, but here they are very large.

This class is used in a few places, and all of them put it on the stack. So let’s see if we can find what thread’s stack this pointer belongs to.

```
0:008> ~*k1

   0  Id: 3700.2404 Suspend: 0 Teb: 000000a9`4d490000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4d3deda8 00007ffd`0d7f43a9     ntdll!ZwWaitForMultipleObjects+0x14

   1  Id: 3700.2728 Suspend: 0 Teb: 000000a9`4d49a000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4daff2a8 00007ffd`0d7f43a9     ntdll!ZwWaitForMultipleObjects+0x14

   2  Id: 3700.1d40 Suspend: 0 Teb: 000000a9`4d49e000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4dcff128 00007ffd`0e5f53dc     win32u!ZwUserMsgWaitForMultipleObjectsEx+0x14

   3  Id: 3700.36b8 Suspend: 0 Teb: 000000a9`4d4a2000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4deff4e8 00007ffd`0d7f43a9     ntdll!ZwWaitForMultipleObjects+0x14

   4  Id: 3700.34d8 Suspend: 0 Teb: 000000a9`4d4a4000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4dfff828 00007ffd`0d7f43a9     ntdll!ZwWaitForMultipleObjects+0x14

   5  Id: 3700.21dc Suspend: 0 Teb: 000000a9`4d4aa000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4e2ff7b8 00007ffd`0e5f53dc     win32u!ZwUserMsgWaitForMultipleObjectsEx+0x14

   6  Id: 3700.2184 Suspend: 0 Teb: 000000a9`4d4ae000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4e4ff768 00007ffd`0ae89910
win32u!ZwGdiDdDDIWaitForVerticalBlankEvent+0x14

   7  Id: 3700.2180 Suspend: 0 Teb: 000000a9`4d4b0000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4ddff7b8 00007ffc`ec21d7e0     win32u!ZwUserMsgWaitForMultipleObjectsEx+0x14

#  8  Id: 3700.381c Suspend: 0 Teb: 000000a9`4d50e000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4d8fd7f8 00007ffd`101dabd1     ntdll!ZwAlpcSendWaitReceivePort+0x14

   9  Id: 3700.3a6c Suspend: 0 Teb: 000000a9`4d524000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4e5ff698 00007ffd`10111b3d     ntdll!ZwWaitForWorkViaWorkerFactory+0x14

  10  Id: 3700.3854 Suspend: 0 Teb: 000000a9`4d52e000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4e0ffac8 00007ffd`0e9501f3     win32u!ZwUserMsgWaitForMultipleObjectsEx+0x14

  11  Id: 3700.3438 Suspend: 0 Teb: 000000a9`4d530000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4e3ff7e8 00007ffd`0e9505f3     win32u!ZwUserMsgWaitForMultipleObjectsEx+0x14

  12  Id: 3700.31bc Suspend: 0 Teb: 000000a9`4d538000 Unfrozen
```

```
Child-SP          RetAddr               Call Site
000000a9`4d7ff838 00007ffd`0e9501f3     win32u!ZwUserMsgWaitForMultipleObjectsEx+0x14

  13  Id: 3700.11e8 Suspend: 0 Teb: 000000a9`4d546000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4e9ff448 00007ffd`0d7f43a9     ntdll!ZwWaitForMultipleObjects+0x14

  14  Id: 3700.5c4 Suspend: 0 Teb: 000000a9`4d552000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4d6ff818 00007ffd`10111b3d     ntdll!ZwWaitForWorkViaWorkerFactory+0x14

  15  Id: 3700.2188 Suspend: 0 Teb: 000000a9`4d554000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4d9ffb38 00007ffd`10111b3d     ntdll!ZwWaitForWorkViaWorkerFactory+0x14

  16  Id: 3700.2df0 Suspend: 0 Teb: 000000a9`4d556000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4dbff4b8 00007ffd`10111b3d     ntdll!ZwWaitForWorkViaWorkerFactory+0x14

  17  Id: 3700.1c74 Suspend: 0 Teb: 000000a9`4d558000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4e1ff3a8 00007ffd`0e5f53dc     win32u!ZwUserMsgWaitForMultipleObjectsEx+0x14

  18  Id: 3700.3190 Suspend: 0 Teb: 000000a9`4d55c000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4e7ff768 00007ffd`0ae89910
win32u!ZwGdiDdDDIWaitForVerticalBlankEvent+0x14

  19  Id: 3700.38b8 Suspend: 0 Teb: 000000a9`4d55e000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4e6ffaa8 00007ffd`10111b3d     ntdll!ZwWaitForWorkViaWorkerFactory+0x14

  20  Id: 3700.3bc4 Suspend: 0 Teb: 000000a9`4d560000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4e8ff878 00007ffd`10111b3d     ntdll!ZwWaitForWorkViaWorkerFactory+0x14

  21  Id: 3700.b34 Suspend: 0 Teb: 000000a9`4d562000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4eaffbf8 00007ffd`10111b3d     ntdll!ZwWaitForWorkViaWorkerFactory+0x14

  22  Id: 3700.2de8 Suspend: 0 Teb: 000000a9`4d564000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4ebffa48 00007ffd`10111b3d     ntdll!ZwWaitForWorkViaWorkerFactory+0x14

  23  Id: 3700.1f70 Suspend: 0 Teb: 000000a9`4d566000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4ecff7c8 00007ffd`10111b3d     ntdll!ZwWaitForWorkViaWorkerFactory+0x14

  24  Id: 3700.1180 Suspend: 0 Teb: 000000a9`4d568000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4efff9f8 00007ffd`10111b3d     ntdll!ZwWaitForWorkViaWorkerFactory+0x14
```

```
  25  Id: 3700.684 Suspend: 0 Teb: 000000a9`4d56a000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4f0ffcd8 00007ffc`ec21d7e0     win32u!ZwUserMsgWaitForMultipleObjectsEx+0x14
```

The `~*k1` command means to select (`~`) all threads (`*`) and execute the `k1` command, which dumps the top frame of the call stack.

I'm not actually interested in the call stacks. What I want to see is the `Child-SP` values, so I can find the thread that `0x000000a9`4e5ff298` belongs to.

And here it is:

```
   9  Id: 3700.3a6c Suspend: 0 Teb: 000000a9`4d524000 Unfrozen
Child-SP          RetAddr               Call Site
000000a9`4e5ff698 00007ffd`10111b3d     ntdll!ZwWaitForWorkViaWorkerFactory+0x14
```

If we look at the thread's call stack, we see

```
0:008> ~9k
Child-SP          RetAddr               Call Site
000000a9`4e5ff698 00007ffd`10111b3d     ntdll!ZwWaitForWorkViaWorkerFactory+0x14
000000a9`4e5ff6a0 00007ffd`0e4e458d     ntdll!TppWorkerThread+0x2fd
000000a9`4e5ff9b0 00007ffd`10157558     kernel32!BaseThreadInitThunk+0x1d
000000a9`4e5ff9e0 00000000`00000000     ntdll!RtlUserThreadStart+0x28
```

This is a thread pool worker thread that is waiting for work. The original work (that created the `FirstWidgetResults`) has evidently already returned, seeing as it's not on the stack any more, and the pointer is in fact below the current stack pointer, so it's in freed stack space.

What we have is a use-after-free of the `FirstWidgetResults`. That explains why it's filled with garbage, and why RAII didn't help: The object did indeed keep the handle valid as long as it was alive, but the object itself is no longer alive!

All uses of this `FirstWidgetResults` class are as automatic variables. Here's an example:

```
using unique_tracker = wil::unique_any<
    TrackerHandle, decltype(&::DestroyTracker), ::DestroyTracker>;

Widget FindWidgetWithEnabledAdapter()
{
  TrackerOptions options;
  /* ... code to set the options ... */

  FirstWidgetResults results;

  // Start the tracker (or die trying)
  unique_tracker tracker;
  THROW_IF_FAILED(CreateTracker(
        &options,
        FirstWidgetResults::OnCallback,
        &results,
        &tracker));

  // Wait for results to arrive
  results.WaitForResults();

  // Fail if no qualifying widget was found
  THROW_HR_IF(HRESULT_FROM_WIN32(ERROR_NOT_FOUND),
              results.GetWidgetId() == INVALID_WIDGET_ID);

  // Stop the tracker and wait for it to be destroyed.
  tracker.reset();
  results.WaitForDestroyed();

  // Return the widget that we found.
  return WidgetFromId(results.GetWidgetId());
}
```

Do you see the problem?

We evaluate the results of the tracker immediately after the `readyEvent` is signaled, and we throw an exception out of the function if we couldn't find an acceptable Widget. Now, the `tracker` variable is inside an RAII type (`unique_tracker`), so it will be destroyed as part of the destruction of locals. However, the early exit means that we never perform the `destroyedEvent.wait()`. We just destruct the `results` without waiting.

Now that we have a theory for the premature destruction, we can look into the error logs to seek confirmation. And indeed, the error logs show that we did throw the `ERROR_NOT_FOUND` because we didn't find any widget.

Okay, now that we have a theory, and the theory is supported by evidence, what can we do to fix the problem? We'll look at that next time.