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

devblogs.microsoft.com/oldnewthing/20240412-00

April 12, 2024

Raymond Chen

A customer ran some stress tests on their program with Application Verifier enabled. Thanks for doing that!

They found that there were rare but repeated crashes where their program appeared to be copying a string from invalid memory due to a pointer that didn't appear to match the member variable it was supposed to have come from. One such hit might be chalked up to a flaky CPU, but they had three, from three different machines.

Since this shows up in stress testing, it's not practical to collect a Time Travel Trace, but let's see what we can do with the crash dumps that were produced.

```
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
(5db8.4180): Access violation - code c0000005 (first/second chance not available)
For analysis of this file, run !analyze -v

rax=000001e081072fe0 rbx=000001e080fe8fc8 rcx=000001e081072fe0
rdx=000001e081070ff0 rsi=000001e081070fe0 rdi=000000000000000f
rip=00007ffc08ab143a rsp=000000b3ed0ffa28 rbp=000001e081072fe0
 r8=0000000000000010  r9=0000000000000000 r10=00007ffc08ab0000
r11=000001e0fa3bfde0 r12=0000000000000000 r13=0000000000000000
r14=000000000000001f r15=000000000000022c
VCRUNTIME140!memcpy+0x12a:
00007ffc`08ab143a movdqu  xmm2,xmmword ptr [rdx+r8-10h] ds:000001e0`81070ff0=????????
0:005> kn
 # RetAddr           Call Site
00 00007ff6`06144342 VCRUNTIME140!memcpy+0x12a
01 (Inline Function) contoso!std::_Char_traits<char,int>::copy+0x13
02 (Inline Function) contoso!std::string::_Construct+0xba
03 (Inline Function) contoso!std::string::{ctor}+0xd3
04 00007ff6`06149dcf contoso!Widget::GetId+0x202
05 00007ff6`0614742d contoso!<lambda_9cd1b560a470dbb5c6e1d5da28bd3866>::operator()+0xc0
09 00007ffc`22785976 ntdll!TppSimplepExecuteCallback+0xa3
0a 00007ffc`2172257d ntdll!TppWorkerThread+0x8f6
0b 00007ffc`227aaa78 kernel32!BaseThreadInitThunk+0x1d
0c 00000000`00000000 ntdll!RtlUserThreadStart+0x28
```

So we have a lambda that called `Widget::GetId`, and we crashed trying to copy a string. Here's what `Widget::GetId` looks like:

```
struct Widget
{
    std::string GetId()
    {
        if (m_uniqueId.empty())
        {
            m_uniqueId = SlowGetId();
        }
        return m_uniqueId; // <<< crash here
    }

private:
    std::string m_uniqueId;

    std::string SlowGetId();
};
```

If we dig into the STL code, we are in the `copy` method at the point where we copy the characters from the string into a newly-allocated buffer.

```
        _CSTD memcpy(_First1, _First2, _Count * sizeof(_Elem));
```

The address we are trying to copy from is in the `rdx` register, `000001e0`81070ff0`, which is presumably the string buffer hiding inside `m_uniqueId`, but when we ask the debugger for the contents of `m_uniqueId`, we see something different:

```
0:005> ?? this->m_uniqueId
class std::string
   +0x000 _Mypair          :
std::_Compressed_pair<std::allocator<char>,std::_String_val<std::_Simple_types<char> >,1>
0:005> ?? this->m_uniqueId._Mypair
class std::_Compressed_pair<std::allocator<char>,std::_String_val<std::_Simple_types<char>
>,1>
   +0x000 _Myval2          : std::_String_val<std::_Simple_types<char> >
0:005> ?? this->m_uniqueId._Mypair._Myval2
class std::_String_val<std::_Simple_types<char> >
   +0x000 _Bx              : std::_String_val<std::_Simple_types<char> >::_Bxty
   +0x010 _Mysize          : ??
   +0x018 _Myres           : 0x10
0:005> ?? this->m_uniqueId._Mypair._Myval2._Bx
union std::_String_val<std::_Simple_types<char> >::_Bxty
   +0x000 _Buf             : [16]  "???"
   +0x000 _Ptr             : 0x000001e0`81074ff0  "fdf551a3ebd7f381"
   +0x000 _Alias           : [16]  "???"
0:005>
```

In memory, the `m_uniqueId` of the widget is a reasonable-looking string of hex digits, and the pointer is nothing like the address we crashed on.

I suspected that at the time we started copying, the string was indeed at `000001e0`81070ff0`, but while we were copying the string, another thread came in and changed the `m_uniqueId`, which freed the string out from under us.

Since this crash occurred when running under Application Verifier, we can ask Application Verifier for the histories of these two memory blocks.

```
0:005> !avrf -?
Verifier package version >= 3.00
Application verifier debugger extension

...
!avrf -hp N           dump last N entries from heap log.
!avrf -hp -a ADDR     searches ADDR in the heap log.
...
```

Great, we can use the `!avrf -hp -a` command to ask AppVerifier to tell us what it knows about an address on the heap.

```
0:005> !avrf -hp -a 0x000001e0`81070ff0

Searching call tracker @ 000001e0fe042fc0 with 422 valid entries ...
--------------------------------------------------------------
2024-04-12T07:00:20.140Z GlobalIndex 19E1E6 ThreadId NA
HeapFree: 1E081070FF0 11 1E0FBB60000 7870

    00007ffc1fae37eb: ucrtbase!_free_base+0x1B
    00007ffb45614ff7: vfbasics!AVrfp_ucrt_free+0x57
    00007ff606144219: contoso!Widget::GetId+0xD9
    00007ff606149400: contoso!<lambda_963292f271044b3ead3564f3abbc4b26>::operator()+0xb7
    00007ffc2279e4a3: ntdll!TppSimplepExecuteCallback+0xA3
    00007ffc22785976: ntdll!TppWorkerThread+0x8F6
    00007ffc2172257d: KERNEL32!BaseThreadInitThunk+0x1D
    00007ffc227aaa78: ntdll!RtlUserThreadStart+0x28

--------------------------------------------------------------
2024-04-12T07:00:20.139Z GlobalIndex 19E1E0 ThreadId NA
HeapAlloc: 1E081070FF0 11 1E0FBB60000 4180

    00007ffc22854438: ntdll!RtlDebugAllocateHeap+0x48
    00007ffc2280d6f0: ntdll!RtlpAllocateHeap+0x7EAB0
    00007ffc2278cd49: ntdll!RtlpAllocateHeapInternal+0x6C9
    00007ffbf8bcc3dc: vrfcore!VfCoreRtlAllocateHeap+0x2C
    00007ffb456137d5: vfbasics!AVrfpRtlAllocateHeap+0x155
    00007ffc1fae1b06: ucrtbase!_malloc_base+0x36
    00007ffb45614e20: vfbasics!AVrfp_ucrt_malloc+0x40
    00007ff6061491f3: contoso!operator new+0x1F
    00007ff606144365: contoso!Widget::SlowGetId+0x25
    00007ff606144185: contoso!Widget::GetId+0x45
    00007ff60614742d: contoso!<lambda_9cd1b560a470dbb5c6e1d5da28bd3866>::operator()+0xc0
    00007ffc22785976: ntdll!TppSimplepExecuteCallback+0xa3
    00007ffc22785976: ntdll!TppWorkerThread+0x8F6
    00007ffc2172257d: KERNEL32!BaseThreadInitThunk+0x1D
    00007ffc227aaa78: ntdll!RtlUserThreadStart+0x28
```

And the history for the string in `m_uniqueId` right now is

```
0:005> !avrf -hp -a 0x000001e0`81074ff0

Searching call tracker @ 000001e0fe042fc0 with 422 valid entries ...
--------------------------------------------------------------
2024-04-12T07:00:20.140Z GlobalIndex 19E1E3 ThreadId NA
HeapAlloc: 1E081074FF0 11 1E0FBB60000 7870

    00007ffc22854438: ntdll!RtlDebugAllocateHeap+0x48
    00007ffc2280d6f0: ntdll!RtlpAllocateHeap+0x7EAB0
    00007ffc2278cd49: ntdll!RtlpAllocateHeapInternal+0x6C9
    00007ffbf8bcc3dc: vrfcore!VfCoreRtlAllocateHeap+0x2C
    00007ffb456137d5: vfbasics!AVrfpRtlAllocateHeap+0x155
    00007ffc1fae1b06: ucrtbase!_malloc_base+0x36
    00007ffb45614e20: vfbasics!AVrfp_ucrt_malloc+0x40
    00007ff6061491f3: contoso!operator new+0x1F
    00007ff606144365: contoso!Widget::SlowGetId+0x25
    00007ff606144185: contoso!Widget::GetId+0x45
    00007ff606149400: contoso!<lambda_963292f271044b3ead3564f3abbc4b26>::operator()+0xb7
    00007ffc2279e4a3: ntdll!TppSimplepExecuteCallback+0xA3
    00007ffc22785976: ntdll!TppWorkerThread+0x8F6
    00007ffc2172257d: KERNEL32!BaseThreadInitThunk+0x1D
    00007ffc227aaa78: ntdll!RtlUserThreadStart+0x28
```

The timestamps and GlobalIndex let us reconstruct the chronology of what happened.

First (GlobalIndex `19E1E0`), the original string was allocated when a lambda called `Widget::GetId` from thread `4180`. Which happens to be the thread we're on right now:

```
0:005> ~.
.  5  Id: 5db8.4180 Suspend: 0 Teb: 000000b3`ecb84000 Unfrozen
      Start: ntdll!TppWorkerThread (00007ffc`22785080)
      Priority: 0  Priority class: 32  Affinity: ff
```

The `GetId()` called `SlowGetId()`, and that's what actually allocated the string.

Next (GlobalIndex `19E1E3`), thread `7870` allocated a replacement string, also through `SlowGetId()`.

And then (GlobalIndex `19E1E6`), thread `7870` freed the first string.

And finally, we crashed because thread `4180` (the current thread) tried to copy from that first string, which was already freed.

My guess is that we had a race condition where two threads called `GetId()` at the same time, and both of them decided to do the lazy initialization of `m_uniqueId`.

Thread `4180` enters `GetId()` and sees that `m_uniqueId` is empty, so it calls `SlowGetId()` to get the ID.

While `SlowGetId()` is doing its slow work, thread `7870` calls `GetId()`, and it too sees that `m_uniqueId` is empty, so it also calls `SlowGetId()` to get the ID.

Thread `4180` finishes getting the ID from `SlowGetId()` and saves it in `m_uniqueId`. It then makes a copy of `m_uniqueId` to return to the caller.

While this copy is being made, thread `7870` finishes its call to `SlowGetId()` and (here's where things go bad) saves it in `m_uniqueId`, which causes the previous string to become freed, even though thread `4180` is still copying from it!

The problem is therefore conflicting multithreaded access to `m_uniqueId`: One thread is reading while another is writing.

The design of the `Widget` class is apparently that multithreaded use is allowed, but simultaneous multithreaded use is permitted only for read operations. You cannot have a write operation concurrent with a read or another write.

The lambdas that called `GetId()` were careful to lock a `shared_mutex` in shared mode, thinking that `GetId()` was a read-only operation. I mean, look at its name: It's "Get". It just reads something!

Unfortunately, the lazy initialization of `m_uniqueId` made `GetId()` a read-write operation, even though its name sure sounds like a read-only operation.

One way to fix this is to make sure all callers of `GetId()` take an exclusive lock rather than a shared lock before calling `GetId()`.

Another way to fix this is to have `GetId()` apply internal locking:

```
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;
}
```