How can we use IsBadWritePtr to fix a buffer overflow, if IsBadWritePtr is itself bad?

devblogs.microsoft.com/oldnewthing/20190315-00

March 15, 2019

Raymond Chen

A customer asked for assistance in investigating an access violation caused by a buffer overflow. They figured that they could probe whether the buffer is large enough to receive the data by using `IsBadWritePtr`, but then they saw that `IsBadXxxPtr` should really be called `CrashProgramRandomly`. They were wondering what alternatives existed to `IsBadXxxPtr`.

The alternative to `IsBadXxxPtr` is *not passing bad pointers in the first place*.

If you are getting an access violation from a buffer overflow, the fix for the problem is not to try to detect the overflow as it happens. the fix is to stop the overflow *before* it happens.

The customer shared their code and the stack trace at which the access violation occurred:

```
msvcrt!memcpy+0xb4
contoso!CBuffer::CopyFromRange+0x92
contoso!CBuffer::ReadAt+0x861
contoso!CLockBytes::ReadAt+0xfd
contoso!CStream::Read+0xe3
contoso!CData::ParseFile+0x606
```

The buffer overflow occurred because the `memcpy` was writing past the end of the buffer passed to `CStream::Read`. The thing to do is not try to detect the maximum writable buffer size, but to stop passing invalid buffer sizes.

Because there's probably writable memory after the buffer that is not part of the buffer. If the invalid buffer size were only slightly larger than the buffer (rather than ridiculously larger than the buffer), you wouldn't have gotten an access violation, but you still had a buffer overflow.

The offending `Read` call came from here:

```
// Code in italics is wrong
    uint32_t numBlocks;
    uint32_t actualBytesRead;

    // First, read the number of blocks.
    HRESULT hr = stream.Read(&numBlocks, sizeof(uint32_t), &actualBytesRead);
    if (FAILED(hr) || actualBytesRead != sizeof(uint32_t)) {
        goto Reject;
    }

    // Next, read the size of each block.
    uint32_t blockSize;
    hr = stream.Read(&blockSize, sizeof(uint32_t), &actualBytesRead);
    if (FAILED(hr) || actualBytesRead != sizeof(uint32_t)) {
        goto Reject;
    }

    // Now read the blocks.
    DWORD i;
    for (i = 0; i < numBlocks; i++)
    {
        // Read each block.
        BLOCK block = { 0 };
        hr = stream.Read(&block, blockSize, &actualBytesRead);
        //               ^^^^^^^^^^^^^^^^ invalid buffer here
        if (FAILED(hr) || actualBytesRead != sizeof(uint32_t)) {
            goto Reject;
        }
```

The stack trace implicates the highlighted line of code.

So how do we prevent the invalid buffer from being passed to the `Read` method?

From code inspection, we see that we read `blockSize` bytes into a `BLOCK` structure, but we didn't take any steps to ensure that `blockSize` is no larger than at `BLOCK`. In other words, we have a buffer of size `sizeof(BLOCK)`, and we ask to read `blockSize` bytes into it, so it is our responsibility to ensure that `blockSize <= sizeof(BLOCK)`.

However, no such buffer size validation was present.

How to fix this depends on how you want to deal with unexpected block sizes.

If your intent is to allow large block sizes and just ignore the fields that are "from the future", then you would read `min(blockSize, sizeof(block))` bytes, and then `Seek` over the extra bytes (if any).

If your intent is to reject large block sizes, then go ahead and reject them.

Raymond Chen

Follow