We're using a smart pointer, so we can't possibly be the source of the leak

devblogs.microsoft.com/oldnewthing/20091119-00

November 19, 2009

Raymond Chen

A customer reported that there was a leak in the shell, and they included the output from Application Verifier as proof. And yup, the memory that was leaked was in fact allocated by the shell:

```
VERIFIER STOP 00000900 : pid 0x3A4: A heap allocation was leaked.
        497D0FC0 : Address of the leaked allocation.
        002DB580 : Adress to the allocation stack trace.
        0D65CFE8 : Address of the owner dll name.
        6F560000 : Base of the owner dll.
1: kd> du 0D65CFE8
0d65cfe8  "SHLWAPI.dll"
1: kd> !heap -p -a 497D0FC0
...
    ntdll!RtlpAllocateHeap+0x0003f236
    ntdll!RtlAllocateHeap+0x0000014f
    Kernel32!LocalAlloc+0x0000007c
    shlwapi!CreateMemStreamEx+0x00000043
    shlwapi!CreateMemStream+0x00000012
    <Unloaded_xyz.dll>+0x000642de
    <Unloaded_xyz.dll>+0x0005e2af
    <Unloaded_xyz.dll>+0x0002d49a
    <Unloaded_xyz.dll>+0x0002a0fd
    <Unloaded_xyz.dll>+0x000289cb
    <Unloaded_xyz.dll>+0x0002a25c
    <Unloaded_xyz.dll>+0x00027225
    <Unloaded_xyz.dll>+0x0002252b
    <Unloaded_xyz.dll>+0x00025394
    <Unloaded_xyz.dll>+0x0004d70f
    Kernel32!BaseThreadInitThunk+0x0000000d
    ntdll!RtlUserThreadStart+0x0000001d
1: kd> dps 002DB580
shlwapi!CreateMemStreamEx+0x43
shlwapi!CreateMemStream+0x12
<Unloaded_xyz.dll>+0x642de
<Unloaded_xyz.dll>+0x5e2af
<Unloaded_xyz.dll>+0x2d49a
<Unloaded_xyz.dll>+0x2a0fd
<Unloaded_xyz.dll>+0x289cb
<Unloaded_xyz.dll>+0x2a25c
<Unloaded_xyz.dll>+0x27225
<Unloaded_xyz.dll>+0x2252b
<Unloaded_xyz.dll>+0x25394
<Unloaded_xyz.dll>+0x4d70f
Kernel32!BaseThreadInitThunk+0xd
ntdll!RtlUserThreadStart+0x1d
```

On the other hand, `SHCreateMemStream` is an object creation function, so it's natural that the function allocate some memory. The responsibility for freeing the memory belongs to the caller.

We suggested that the customer appears to have leaked the interface pointer. Perhaps there's a hole where they called `AddRef` and managed to avoid the matching `Release`.

“Oh no,” the customer replied, “that’s not possible. We call this function in only one place, and we use a smart pointer, so a leak is impossible.” The customer was kind enough to include a code snippet and even highlighted the lines that proved they weren’t leaking.

```
CComPtr<IStream> pMemoryStream;
CComPtr<IXmlReader> pReader;
UINT nDepth = 0;
//Open read-only input stream
pMemoryStream = ::SHCreateMemStream(utf8Xml, cbUtf8Xml);
```

The exercise for today is to identify the irony in the highlighted lines.

Hint. Answers (and more discussion) tomorrow.

Raymond Chen

Follow