If you get a procedure address by ordinal, you had better be absolutely sure it's there, because the failure mode is usually indistinguishable from success

devblogs.microsoft.com/oldnewthing/20141210-00

December 10, 2014

Raymond Chen

A customer reported that the `GetProcAddress` function was behaving strangely.

> We have this code in one of our tests:
>
> ```
> typedef int (CALLBACK *T_FOO)(int);
> void TestFunctionFoo(HINSTANCE hDLL)
> {
>   // Function Foo is ordinal 1 in our DLL
>   T_FOO pfnFoo = (T_FOO)GetProcAddress(hDLL, (PCSTR)1);
>   if (pfnFoo) {
>     ... run tests on pfnFoo ...
>   }
> }
> ```
>
> Recently, this test started failing in bizarre ways. When we stepped through the code, we discovered that `pfnFoo` ends up calling `Bar` instead of `Foo`. The first time we try to test `pfnFoo`, we get stack corruption because `Bar` has a different function prototype from `Foo`, and of course on top of that the test fails horribly because it's calling the wrong function!
>
> When trying to narrow the problem, we found that the issue began when the test was run against a version of the DLL that was missing the `Foo` function entirely. The line
>
> ```
> Foo @1
> ```
>
> was removed from the DEF file. Why did the call to `GetProcAddress` succeed and return the wrong function? We expected it to fail.

Let's first consider the case where a DLL exports no functions by ordinal.

```
EXPORTS
    Foo
    Bar
    Plugh
```

The linker builds a list of all the exported functions (in an unspecified order) and fills in two arrays based on that list. If you look in the DLL image, you'll see something like this:

```
Exported Function Table
00049180 address of Bar
00049184 address of Foo
0004918C address of Plugh
Exported Names
00049190 address of the string "Bar"
00049194 address of the string "Foo"
00049198 address of the string "Plugh"
```

There are two parallel arrays, one with function addresses and one with function names. The string `"Bar"` is the first entry in the exported names table, and the function `Bar` is the first entry in the exported function table. In general, the string in the *N*th entry in the exported names table corresponds to the function in the *N*th entry of the exported function table.

Since it is only the relative position that matters, let's replace the addresses with indices.

```
Exported Function Table
[1] address of Bar
[2] address of Foo
[3] address of Plugh
Exported Names
[1] address of the string "Bar"
[2] address of the string "Foo"
[3] address of the string "Plugh"
```

Okay, now let's introduce functions exported by ordinal. When you do that, you're telling the linker, "Make sure this function goes into the *NN*th slot in the exported function table." Suppose your DEF file went like this:

```
EXPORTS
    Foo @1
    Bar
    Plugh
```

This says "First thing we do is put `Foo` in slot 1. Once that's done, fill in the rest arbitrarily."

The linker says, "Okay, I have a total of three functions, so let me build two tables with three entries each."

```
Exported Function Table
[1] address of ?
[2] address of ?
[3] address of ?
Exported Names
[1] address of ?
[2] address of ?
[3] address of ?
```

"Now I place `Foo` in slot 1."

```
Exported Function Table
[1] address of Foo
[2] address of ?
[3] address of ?
Exported Names
[1] address of the string "Foo"
[2] address of ?
[3] address of ?
```

"Now I fill in the rest arbitrarily."

```
Exported Function Table
[1] address of Foo
[2] address of Bar
[3] address of Plugh
Exported Names
[1] address of the string "Foo"
[2] address of the string "Bar"
[3] address of the string "Plugh"
```

Since you explicitly placed `Foo` in slot 1, when you do a `GetProcAddress(hDLL, 1)`, you will get `Foo`. On the other hand, if you do a `GetProcAddress(hDLL, 2)`, you will get `Bar`, or at least you will with this build. With the next build, you may get something else, because the linker just fills in the slots arbitrarily, and next time, it may choose to fill them arbitrarily in some other order. Furthermore, if you do a `GetProcAddress(hDLL, 6)`, you will get `NULL` because the table has only three functions in it.

I hope you see where this is going.

If you delete `Foo` from the `EXPORTS` section, this stops exporting `Foo` but says nothing about what goes into slot 1. As a result, the linker is free to put anything it wants into that slot.

```
Exported Function Table
[1] address of Bar
[2] address of Plugh
Exported Names
[1] address of the string "Bar"
[2] address of the string "Plugh"
```

Now, when you do a `GetProcAddress(hDLL, 1)`, you get `Bar`, since that's the function that happened to fall into slot 1 this time.

The moral of the story is that if you try to obtain a function by ordinal, then it had better be there, because there is no reliable way of being sure that the function you got is one that was explicitly placed there, as opposed to some other function that happened to be assigned that slot arbitrarily.

Related reading: How are DLL functions exported in 32-bit Windows?

Raymond Chen

Follow