So how bad is it that I'm calling RegOpenKey instead of RegOpenKeyEx?

devblogs.microsoft.com/oldnewthing/20160120-00

January 20, 2016

Raymond Chen

A customer had some code that called the `RegOpenKey` function and was concerned by the remark in MSDN:

> **Note** This function is provided only for compatibility with 16-bit versions of Windows. Applications should use the **RegOpenKeyEx** function.

What are the dire consequences of using this old function instead of the new one?

In general, not much.

If you call `RegOpenKey`, then some compatibility stuff kicks in, and then it goes ahead and behaves as if you had called `RegOpenKeyEx`.

In the specific case of `RegOpenKey`, the compatibility stuff is mentioned in the parameter documentation of `RegOpenKey`:

> *lpSubKey*: If this parameter is **NULL** or a pointer to an empty string, the function returns the same handle that was passed in.

This is different from `RegOpenKeyEx`, which always returns a new key. It means that if you pass `NULL` as the `lpSubKey`, then the returned registry key is the same as the one that you passed in, and therefore it does *not* create a new obligation to call `RegCloseKey`. In other words, this code has a potential bug:

```
void DoSomething(HKEY hkey, PCSTR subkeyName)
{
  HKEY subkey;
  if (RegOpenKey(hkey, subkeyName, &subkey) == ERROR_SUCCESS) {
    // do something
    RegCloseKey(subkey);
  }
}
```

The bug occurs if `subkeyName` is `NULL` or `""`. In that case, the special 16-bit compatibility behavior kicks in, and `subkey` is set to a copy of `hkey`. This means that when you do `RegCloseKey(subkey)`, you are *closing the original* `hkey`, and the caller will probably be rather upset that you closed a key out from under it.

If you know that `subkeyName` is never `NULL` or `""`, then you can safely close the key. Otherwise, you either need to check against this special case or (better) just switch to `RegOpenKeyEx` so you don't have to deal with the special case in the first place.

Raymond Chen

Follow