# It is not unreasonable to expect uninitialized garbage to change at any time, you don't need to ask for an explanation

devblogs.microsoft.com/oldnewthing/20111123-00

November 23, 2011

Raymond Chen

A customer admitted that they had a bug in their code:

```
#define UNICODE
#define _UNICODE
#include <windows.h>
// error checking removed for expository purposes
// code that writes out the data
RegSetValueEx(hkey, pszValue, 0, REG_SZ, (const BYTE *)pszData,
              _tcslen(pszData) * sizeof(TCHAR) + 1);
// code that reads the data
DWORD dwType, cbData;
RegQueryValueEx(hkey, pszValue, NULL, &dwType, NULL, &cbData);
TCHAR *pszData = new TCHAR[cbData / sizeof(TCHAR)];
RegQueryValueEx(hkey, pszValue, NULL, &dwType, pszData, &cbData);
```

One bug in the above code is in the final parameter passed to `RegSetValueEx`: It's supposed to be the count in bytes, but the calculation appends only one byte for the terminating null instead of a full `TCHAR`. In other words, it should be

```
RegSetValueEx(hkey, pszValue, 0, REG_SZ, (const BYTE *)pszData,
              _tcslen(pszData) * sizeof(TCHAR) + sizeof(TCHAR));
```

For concreteness, let's say the original string was five `TCHAR`s in length, not counting the terminating null. Therefore, the correct buffer size is 12 bytes, but they passed only 11 to `RegSetValueEx`.

This error is compounded in the code that reads the value back: The code happily divides `cbData / sizeof(TCHAR)` without checking that the division is even. In our example, the call returns a length of 11 bytes. They divide by `sizeof(TCHAR)` (which is 2, since the code is compiled as Unicode), leaving 5 (remainder discarded), causing them to allocate a 5-`TCHAR` buffer.

That error would have been okay by itself except for another error, which is calling `RegQueryValueEx` a second time with an invalid buffer size: The `cbData` variable remains the original value of 11 even though they allocated only 10 bytes. The subsequent `RegQueryValueEx` call reads 11 bytes into a 10-byte buffer.

The customer conceded that the code that writes the value is buggy, but points out that the code "worked" on Windows XP, in the sense that the string read back from the registry was correct. But Windows Vista "broke" their program, because the string read back now contained garbage at the end. Instead of returning `"Hello"`, it returned `"HelloÈ┼۞"`. The customer wanted to know what change to Windows Vista broke their program.

The change to Windows Vista that broke their program is known as "luck running out." The program contained three bugs, which combined to form a heap buffer write overflow. The uninitialized garbage at the end of the heap block they allocated happened to be zero on Windows XP due to a coincidence in the way their program allocated and freed memory. Consequently, when the data was read from the registry, the "string" ended in a single null byte instead of two. The extra null byte that "happened to be there already" combined with the single null byte read from the registry to form a proper null terminator.

When run on Windows Vista, that happy coincidence no longer took place, and the uninitialized garbage was nonzero, resulting in the subsequent attempt to use the string to read past the end of the buffer and pick up heap garbage. (Yay, bug number four: read overflow.) Why was the uninitialized garbage different?

It's different because there was nothing forcing it to be the same. The internals of the heap manager change all the time. (Look-aside lists, low fragmentation heap, and fault-tolerant heap are just a few recent examples.) Any of these changes will result in heap memory being used and reused differently. Plus, changes in other parts of Windows may have allocated and freed memory differently between Windows XP and Windows Vista. Heck, the program itself may have allocated and freed memory differently due to the change in operating system. (For one thing, the length of the string `"Windows Vista"` is different from the length of the string `"Windows XP"`.)

Uninitialized garbage will contain unpredictable values. There's no point asking why you got a different unpredictable value this time.

Raymond Chen

**Follow**