# How can code that tries to prevent a buffer overflow end up causing one?

devblogs.microsoft.com/oldnewthing/20050107-00

January 7, 2005

Raymond Chen

If you read your language specification, you'll find that the `...ncpy` functions have extremely strange semantics.

> The **strncpy** function copies the initial count characters of *strSource* to *strDest* and returns *strDest*. If *count* is less than or equal to the length of *strSource*, a null character is not appended automatically to the copied string. If *count* is greater than the length of *strSource*, the destination string is padded with null characters up to length *count*.

In pictures, here's what happens in various string copying scenarios.

| `strncpy(strDest, strSrc, 5)` | | |
|---|---|---|
| `strSource` | W e l c o m e \0 | |
| `strDest` | W e l c o | observe no null terminator |

| `strncpy(strDest, strSrc, 5)` | | |
|---|---|---|
| `strSource` | H e l l o \0 | |
| `strDest` | H e l l o | observe no null terminator |

| | | | | | | |
|---|---|---|---|---|---|---|
| `strncpy(strDest, strSrc, 5)` | | | | | | |
| `strSource` | H | i | \0 | | | |
| `strDest` | H | i | \0 | \0 | \0 | observe null padding to end of strDest |

Why do these functions have such strange behavior?

Go back to the early days of UNIX. Personally, I only go back as far as System V. In System V, file names could be up to 14 characters long. Anything longer was truncated to 14. And the field for storing the file name was exactly 14 characters. Not 15. The null terminator was implied. This saved one byte.

Here are some file names and their corresponding directory entries:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `passwd` | p | a | s | s | w | d | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 |
| `newsgroups.old` | n | e | w | s | g | r | o | u | p | s | . | o | l | d |
| `newsgroups.old.backup` | n | e | w | s | g | r | o | u | p | s | . | o | l | d |

Notice that `newsgroups.old` and `newsgroups.old.backup` are actually the same file name, due to truncation. The too-long name was silently truncated; no error was raised. This has historically been the source of unintended data loss bugs.

The `strncpy` function was used by the file system to store the file name into the directory entry. This explains one part of the odd behavior of `strcpy`, namely why it does not null-terminate when the destination fills. The null terminator was implied by the end of the array. (It also explains the silent file name truncation behavior.)

But why null-pad short file names?

Because that makes scanning for file names faster. If you guarantee that all the “garbage bytes” are null, then you can use `memcmp` to compare them.

For compatibility reasons, the C language committee decided to carry forward this quirky behavior of `strncpy`.

So what about the title of this entry? How did code that tried to prevent a buffer overflow end up causing one?

Here’s one example. (Sadly I don’t read Japanese, so I am operating only from the code.) Observe that it uses `_tcsncpy` to fill the `lpstrFile` and `lpstrFileTitle`, being careful not to overflow the buffers. That’s great, but it also leaves off the null terminator if the string is too long. The caller may very well copy the result out of that buffer to a second buffer. But the `lstrFile` buffer lacks a proper null terminator and therefore exceeds the length the caller specified. Result: Second buffer overflows.

Here’s another example. Observe that the function uses `_tcsncpy` to copy the result into the output buffer. This author was mindful of the quirky behavior of the `strncpy` family of functions and manually slapped a null terminator in at the end of the buffer.

But what if `ccTextMax` = 0? Then the attempt to force a null terminator dereferences past the beginning of the array and corrupts a random character.

What’s the conclusion of all this? Personally, my conclusion is simply to avoid `strncpy` and all its friends if you are dealing with null-terminated strings. Despite the “str” in the name, these functions do **not** produce null-terminated strings. They convert a null-terminated string into a raw character buffer. Using them where a null-terminated string is expected as the second buffer is plain wrong. Not only do you fail to get proper null termination if the source is too long, but if the source is short you get unnecessary null padding.

Raymond Chen

**Follow**