What happens if you pass a source length greater than the actual string length?

devblogs.microsoft.com/oldnewthing/20070919-00

September 19, 2007

Raymond Chen

Many functions accept a source string that consists of both a pointer and a length. And if you pass a length that is greater than the length of the string, the result depends on the function itself.

Some of those functions, when given a string and a length, will stop either when the length is exhausted or a null terminator is reached *whichever comes first*. For example, if you pass a `cchSrc` greater than the length of the string to the `StringCchCopyN` function, it will stop at the null terminator.

On the other hand, many other functions (particularly those in the NLS family) *will cheerfully operate past a null character if you ask them to*. The idea here is that since you passed an explicit size, you're consciously operating on a buffer which might contain embedded null characters. Because, after all, if you passed an explicit source size, you really meant it, right? (Maybe you're operating on a `BSTR`, which supports embedded nulls; to get the size of a `BSTR` you must use a function like `SysStringLen`.) For example, if you call `CharUpperBuff(psz, 20)`, then the function really will convert to uppercase 20 `TCHAR`s starting at `psz`. If there happens to be a null character at `psz[10]`, the function will convert the null to uppercase and continue converting the next ten `TCHAR`s as well.

I've seen programs crash because they thought that functions like `CharUpperBuff` and `MultiByteToWideChar` stopped when they encountered a null. For example, somebody might write

```
// buggy code - see discussion
void someFunction(char *pszFile)
{
 CharUpperBuff(pszFile, MAX_PATH);
 ... do something with pszFile ...
}
void Caller()
{
 char buffer[80];
 sprintf(buffer, "file%d", get_fileNumber());
 someFunction(buffer);
}
```

The intent here was for `someFunction` to convert the string to uppercase before operating on it, up to `MAX_PATH` characters' worth, but instead what happens is that the `MAX_PATH` characters starting at `pszFile` are converted, even though the actual buffer is shorter! As a result, `MAX_PATH` – 80 = 220 characters beyond the end of `buffer` are also converted. And since that's a stack buffer, those bytes are likely to include the return address. Result: Crash-o-rama. Things get even more interesting if the short buffer had been allocated on the heap instead. Then instead of corrupting your return address (which you would probably notice as soon as the function returned), you corrupt the heap, which typically doesn't manifest itself in a crash until long after the offending function has left the scene of the crime.

Critique, then, this replacement function:

```
//  buggy code - do not use
int invariant_strnicmp(char *s1, char *s2, size_t n)
{
 // [Update: 9:30am - typo fixed]
 return CompareStringA(LOCALE_INVARIANT, NORM_IGNORECASE,
                       s1, n, s2, n) - CSTR_EQUAL;
}
```

(Michael Kaplan has one answer different from the one I was looking for.)

Raymond Chen

Follow