# If you say that your buffer can hold 200 characters, then it had better hold 200 characters

devblogs.microsoft.com/oldnewthing/20180523-00

May 23, 2018

Raymond Chen

A security vulnerability report claimed that there was a vulnerability in the `GetDoodadName` function (not the actual function name):

> There is a buffer overflow bug in the `GetDoodadName` function. If the doodad's name is 10 characters long, and the caller provides a buffer of size 11 characters, but specifies a buffer size of 200, then the `GetDoodadName` function will write more than 11 characters (10 characters for the name, plus the null terminator). Even though the caller passed an incorrect buffer size, the overflow should not happen because the caller's buffer was large enough to hold the actual result.

The original report was difficult to understand, probably because English was not the finder's native language, and there are parts of the report where I couldn't figure out what the finder was trying to say.

Going back to the issue at hand: If you pass a buffer to a function and say that it can hold up to 200 characters, then the function is welcome to use the entire buffer, even if the final result doesn't require full use of the buffer. This is just part of the basic ground rules for programming:

> A function is permitted to write to the full extent of the buffer provided by the caller, even if not all of the buffer is required to hold the result.

In this case, what happens is that the `GetDoodadName` function relies upon an internal function, let's call it `GetDoodadFullName`, which returns a fully-qualified name. It then removes the unnecessary qualifications, resulting in the final doodad name for the caller.

The simple implementation of this would go something like this:

```
DWORD GetDoodadName(
    HANDLE doodad,
    PWSTR buffer, UINT bufferSize,
    UINT* actualSize)
{
 *actualSize = 0;

 UINT actualFullSize;
 DWORD result = GetDoodadFullName(doodad,
   nullptr, 0, &actualFullSize);

 // If something went wrong other than "buffer too small",
 // then give up.
 if (result != ERROR_MORE_DATA) return result;

 PWSTR fullName = (PWSTR)HeapAlloc(
    GetProcessHeap(), 0, actualFullSize);
 if (!fullName) return ERROR_NOT_ENOUGH_MEMORY;
 result = GetDoodadFullName(doodad,
    fullName, actualFullSize, &actualFullSize);
 if (result == ERROR_SUCCESS) {
  *actualSize = ExtractLocalNameFromFullName(
    fullName, buffer, bufferSize);
 }
 HeapFree(GetProcessHeap(), 0, fullName);
 return result;
}
```

Since you're going to have to call `GetDoodadFullName` anyway, you may as well see if you're lucky, and the full name fits inside the caller-provided buffer. In that case, you need only call `GetDoodadFullName` once, and you don't need to allocate the temporary buffer either. That saves you a memory allocation and two calls to the doodad server.

```
DWORD GetDoodadName(HANDLE doodad, PWSTR buffer, UINT bufferSize,
    UINT* actualSize)
{
 *actualSize = 0;

 UINT actualFullSize;
 DWORD result = GetDoodadFullName(doodad,
    buffer, bufferSize, &actualFullSize);
 if (result == ERROR_SUCCESS) {
  // The caller's buffer is big enough to hold the full name.
  *actualSize = ExtractLocalNameFromFullName(
    buffer, buffer, bufferSize);
  return result;
 }

 // If something went wrong other than "buffer too small",
 // then give up.
 if (result != ERROR_MORE_DATA) return result;

 PWSTR fullName = (PWSTR)HeapAlloc(
    GetProcessHeap(), 0, actualFullSize);
 if (!fullName) return ERROR_NOT_ENOUGH_MEMORY;
 result = GetDoodadFullName(doodad,
    fullName, actualFullSize, &actualFullSize);
 if (result == ERROR_SUCCESS) {
  *actualSize = ExtractLocalNameFromFullName(
    fullName, buffer, bufferSize);
 }
 HeapFree(GetProcessHeap(), 0, fullName);
 return result;
}
```

This is a legitimate optimization because the function has free use of the caller-provided buffer, for the full extent of the caller-specified size of the buffer, until the function returns. And this function takes advantage of this freedom by using the caller-provided buffer as a temporary buffer for holding the full name.

If the caller provides a buffer of size 200, then that buffer had better be 200 characters in size, and all 200 of those characters had better be expendable.

What's even more dangerous about this is that the caller cannot guarantee the length of the doodad's local name. A doodad's name can be changed by anybody who can find the doodad in the system doodad table, so you can't say, "Well, my code created the doodad with a local name whose length I know to be exactly 10, so it's safe to overstate the buffer size because I know that the result won't use more than 11 characters." Some other program may have changed the doodad's name, and your "knowledge" that the result will require only 11 of those characters is no longer valid.

Curiously, the finder even acknowledged the fact that the name could change for reasons outside the program's control, and noted that if the new name requires more than 11 characters, then more than 11 characters will be modified.

So I don't know what the finder was trying to say. Since the length of the name cannot be known ahead of time, the caller doesn't know how much of the putatively 200-character buffer will be used, so the caller needs to be prepared for the case that *all of it* will be used. If the caller had important data at character 12, the caller may be in for an unpleasant surprise.

The buffer overflow in the report is not a vulnerability in the `GetDoodadName` function. It is a vulnerability in the caller, for passing the wrong buffer size.

We asked the finder to clarify why they considered this a flaw in the `GetDoodadName` function, but the response was not readily comprehensible. They seemed to be more interested in the change in behavior when the incorrectly-specified buffer size is large enough to hold the full name, as opposed to when it isn't.

One part that was sort of understandable went like this (after correcting grammar):

> According to the design principle, even if I have provided an incorrect buffer size, a crash should not happen, because the user provided an actual buffer large enough to hold the (undecorated) doodad name.

I'm not sure what design principle says that if a caller provides an incorrect buffer size, we should somehow detect and avoid overflowing the buffer. If that were possible, then why have buffer size parameters at all? Just detect the correct buffer size automatically for all callers!

The changing or missing antecedents makes the clarification hard to decipher as well. Sometimes the invalid parameter came from "me", sometimes it was provided by "the user", and sometimes the agent that crashed is left unspecified.

The crash is interesting if it occurs at a different security level from the caller who passed invalid (or malicious) parameters. In this case, the invalid parameters are coming from the calling application, and the buffer overflow occurs in the calling application, and the crash occurs in the calling application. So everything happens at the same security level, and there is no elevation. What you found is a way for a malicious caller to corrupt its own memory in a very roundabout way.

Raymond Chen

**Follow**