The CoInitializeSecurity function demands an absolute security descriptor

devblogs.microsoft.com/oldnewthing/20240902-00

September 2, 2024

One unfortunate requirement of the `CoInitializeSecurity` function is that it requires a security descriptor in absolute format. I call this unfortunate because the most common format for security descriptors is self-relative format. A common way to specify a security descriptor is to use the security descriptor definition language (SDDL), and the `ConvertStringSecurityDescriptorToSecurityDescriptor` function that converts these strings to security descriptors produces self-relative security descriptors. Another common way to specify a security descriptor is as a block of bytes, perhaps stored in the registry, or stored in a file, or just hard-coded into a binary blob, and these are naturally in self-relative format, since absolute format would be dependent on the location of the block of bytes in memory.

Unfortunately, if you pass a self-relative security descriptor to `CoInitializeSecurity` it fails with `HRESULT_FROM_WIN32(ERROR_BAD_DESCRIPTOR_FORMAT)`. The requirement that the security descriptor be in absolute format is documented, but it's still annoying.

Internally, the reason is that the `CoInitializeSecurity` converts the incoming security descriptor to relative format by calling `MakeSelfRelativeSD`, without first checking whether the conversion is even necessary. The `MakeSelfRelativeSD` function fails with `ERROR_BAD_DESCRIPTOR_FORMAT` if the security descriptor is not absolute, and that error is propagated from `CoInitializeSecurity`.

Now, `CoInitializeSecurity` could be updated to support self-relative security descriptor. It would just be a check whether the security descriptor is already self-relative, and using the existing one if so. However, this would create a compatibility problem, but not in the way you're used to thinking of them.

Most of the time, the compatibility issue is "Will existing old code continue to work on a new system?" But in this case, the compatibility issues goes the other way: "Will new code continue to work on an old system?"

If `CoInitializeSecurity` started allowing self-relative security descriptors, then somebody writing code today could take advantage of this new feature (perhaps unwittingly), and then encounter problems when their program is run on an older version of Windows. There is no obvious indication as to what went wrong because the function `CoInitializeSecurity` does exist on the old system.

The scenario is that somebody wants to be compatible with (say) Windows 10 Version 1803, so they compile their program with the Windows 10 Version 1803 SDK, and it compiles just fine. It also runs fine on Windows 11, and since they used the Windows 10 version 1803 SDK, they erroneously conclude that the program also works on Windows 10 version 1803.

Now, maybe you say that somebody who does this deserves what they get, and maybe you're right, but Windows traditionally has been wary of creating too many of these "pits of failure" where somebody proceeding with the best of intentions can stumble into a bad situation.