Why does canonical order for ACEs put deny ACEs ahead of allow ACEs?

devblogs.microsoft.com/oldnewthing/20070608-00

June 8, 2007

Raymond Chen

So-called canonical order for ACEs in an access control list places deny ACEs ahead of allow ACEs. Why is this the canonical order? Because it gives results that are sensible. The algorithm for determining whether a user has access to an object protected by an ACL is as follows:

```
let access-still-needed = access-requested
for each ACE in the ACL that applies to the user (in order)
    if it is a deny ACE:
        if (access-still-needed & ace-mask) return access-denied
    if it is an allow ACE:
        access-still-needed &= ~ace-mask
end for loop
if access-still-needed != 0 return access-denied
return access-granted
```

In words, we go through the ACEs in the ACL in the order they appear, paying attention only to the ones that apply to the user, i.e. the ones whose SIDs are present in the user's token. If a permission is being denied, and the user is still looking for that permission, then access is denied. If a permission is being granted, then those permissions are subtracted from the permissions the user is still looking for. If, at the end of the day, all the permissions the user requests have been granted, then access is granted. The key detail in the above algorithm is that deny ACEs apply to permissions *not yet granted* and not to the original set of permissions requested. If you deny write, but an earlier ACE grants it, then the deny has no effect. Let's look at what happens if we apply this algorithm to an ACL that is not in canonical order. Our ACL is as follows:

- Grant write access to Alice.
- Deny read and write access to Users.
- Grant read access to Users.

Let’s say that Alice wants write access. We start with access-still-needed = write, and the first ACE grants it, leaving access-still-needed equal to zero. The second ACE denies read and write, but Alice already got write access thanks to the first rule, and she never asked for read access, so this deny ACE has no effect. The third ACE also has no effect since Alice wasn’t looking for read access. Result: Alice gets write access. On the other hand, suppose Alice wants read access. The first ACE has no effect, since Alice isn’t interested in write access. The second ACE then denies access since Alice is being denied read access which she hasn’t gotten yet. Alice’s request is rejected without even looking at the third ACE. Notice that if the ACEs are not canonically-ordered, you can’t use a simple rule like “deny ACEs take priority over allow ACEs”. The rule is “Well, you have to go through each ACE one by one, and you get access if you get all the things you want before somebody denies them.” It sort of turns into a game show. Since graphical ACL editors typically don’t show the order of the ACEs, some sort of canonical order needs to be established so that you don’t run afoul of this “order of operations” problem. Notice that in the algorithm above, you can swap two adjacent allow ACEs and two adjacent deny ACEs without affecting the result, but you cannot swap the positions of an allow and a deny ACE. Therefore, the canonical ordering must either be “all deny ACEs come before all allow ACEs” or “all allow ACEs come before all deny ACEs”. Note, however, that if you choose to have all allow ACEs come before all deny ACEs, then you don’t need deny ACEs at all! If you look at the algorithm above, if there is no ACE that mentions the permission you want, then access is denied. The deny ACEs don’t add anything to the picture:

```
// assuming that all allow ACEs come before all deny ACEs
let access-still-needed = access-requested
for each allow ACE in the ACL that applies to the user (in order)
    access-still-needed &= ~ace-mask
end for loop
for each deny ACE in the ACL that applies to the user (in order)
    if (access-still-needed & ace-mask) return access-denied
end for loop
if access-still-needed != 0 return access-denied
return access-granted
```

Notice that once you make it out of the first “for” loop, the return value is going to be access-denied if access-still-needed is nonzero. All the deny ACEs give you is another way to say “no”. But you were going to say “no” anyway. Therefore, for deny ACEs to be meaningful, the canonical ordering should place them ahead of allow ACEs. That way, you get three tiers of permission instead of just two:

- If there is a deny ACE, then it is denied.
- If there is no deny ACE but there is an allow ACE, then it is allowed.
- If there is neither a deny ACE or an allow ACE, then it is denied.

Postscript: Our sample non-canonical ACL above can easily be converted to an equivalent canonical one:

> Grant write access to Alice.

Why does this work? Well, first notice that the second rule ("Deny read and write access to Users") completely overrides the third rule ("Grant read access to Users"), since any attempt by the third rule to grant read access to Users will be thwarted by the second rule, which denies it.

But the second rule itself is unnecessary. We are taking advantage of the test outside the loop in the access algorithm: `if access-still-needed != 0 return access-denied`. This rule means that the default for all access mode is to deny. Therefore, you don't need to deny anything explicitly unless you have a broader rule later that grants it. (If you have a more narrow rule later that grants it, then that narrower rule is pointless, as we saw in the previous paragraph.) In other words, there's no point denying read and write to Users since merely not saying anything is equivalent to a denial.

Raymond Chen

Follow