Hey there token, long time no see! (Did you do something with your hair?)

devblogs.microsoft.com/oldnewthing/20100910-00

September 10, 2010

Raymond Chen

Consider a system where you have a lot of secured objects, and suppose further that checking whether a user has access to an object is a slow operation. This is not as rare as you might think: Even though a single access check against a security descriptor with a small number of ACEs might be fast, you can have objects with complicated security descriptors or (more likely) users who belong to hundreds or thousands of security groups. Since checking whether a security descriptor grants access to a token is potentially[1] $O(nm)$ in the number of ACEs in the security descriptor and the number of groups the user belongs to (since each ACE needs to be checked to see if it matches each group), even a check against a small security descriptor can multiply out to a slow operation when the user belongs to thousands of groups. Suppose your profiling shows that you spend a lot of time checking tokens against security descriptors. How can you create a cache of access/no-access results so you can short-circuit the expensive security check when a user requests access to an object? (And obviously, you can't have any false positives or false negatives. Security is at stake here!) First, let's look at things that don't solve the problem: One option is to query the SID from the token and cache the access/no-access result with the SID. This option is flawed because between the two checks, the user's group membership may have changed. For example, suppose object X is accessible to members of Group G. Bob starts out as a member of Group G, asks you for access, and you grant it and cache the fact that *Bob has access to object X*. Later that day, Bob's membership in Group G is revoked, and when Bob logs on the next day, his token won't include Group G. If you had merely cached Bob's SID, you would have seen the entry in the cache and said, "Welcome back, Bob. Have fun with object X!" Bob then rubs his hands together and mutters *Excellent!* and starts making unauthorized changes to object X. Now, Bob's membership in Group G might have been revoked at Bob's request. Reducing one's privileges is a common safety measure. For example, Bob might remove his membership in the Administrators group so he won't accidentally delete an important file. Low Rights Internet Explorer intentionally removes a slew of privileges from its token so that the scope of damage of an attack from a malicious site is limited. Okay, so how can we recognize that the Bob that comes back has different group membership from the Bob that visited us the first time? You can do this with the help of the `TOKEN_STATISTICS` structure. This structure contains a number of locally-unique values which can be used to recognize and correlate tokens. A locally-unique value is a value that is unique on the local machine until the operating system is shut down or restarted. You request the statistics for a token by calling `GetTokenInformation` function, passing `TokenStatistics` as the information class. The `AuthenticationId` is known in some places as the `LogonId` because it is assigned to the logon session that the access token represents. There can be many tokens representing a single logon session, so that won't work for our purposes. The `TokenId` is a little closer. It is a locally-unique value assigned to a token when it is created. This value remains attached to that token until it is destroyed. This is closer, but still not perfect, because Bob can enable or disable privileges, and that doesn't change the token, but it sure changes the result of a security check! The `ModifiedId` is a value which is updated each time a token is modified. Therefore, when you want to cache that *This particular token has access to this security descriptor*, you should use the `ModifiedId` as the key. (Remember, locally-unique values are good only until the system shuts down or restarts, so don't cache them across reboots!) Now, a cache with a bad policy is another name for a memory leak, so be careful how much and how long you cache the results of previous security checks. You don't want somebody who goes into a loop alternatively calling `AdjustTokenPrivileges` and your function to cause your cache to consume all the memory in the system. (Each call to `AdjustTokenPrivileges` updates the `ModifiedId`, which causes your code to create a new cache entry.) Now, you might decide to use as your lookup key the `ModifiedId` and some unique identifier associated with the object. This means that if Bob accesses 500 objects, you have 500 cache entries saying *Bob has access to object 1* through *Bob has access to object 500*. (And you have to remembering to purge all cached results for an object if the object's security descriptor changes.) It turns out you can do better. Even though you may have millions of objects, you probably don't have millions of security descriptors. For example, consider your hard drive: Most of the files on that hard drive use one of just a handful of security descriptors. In particular, it's nearly always the case that all files in a directory share the same security descriptor, because they start out with the security descriptor inherited from the directory, and most people don't bother customizing it. Even if your hard drive is on a server with hundreds of users connecting and creating files, you will probably only have a few thousand unique security descriptors. A better cache key would be the `ModifiedId` of the token being checked and the self-relative security descriptor that the token was checked against. If Bob accesses 500 objects, there will probably be only around five unique security descriptors. That's only five cache entries for Bob. It also saves you the trouble of remember to purge the cache when an object's security descriptor changes, since a new security descriptor changes one of the lookup keys, so it gets a new cache entry. Since security descriptors tend to be shared among many objects, you get two bonus benefits: The old security descriptor is probably still being used by some other object, so you may as well leave it in the cache and let it age out naturally. And second, there's a good chance the new security descriptor is already in your cache because it's probably already being used by some other object. [1]I use the word *potentially* because Windows Vista introduced an optimization which preprocesses the token to reduce the complexity of the access check operation. In practice, the access check is linear in the number of ACEs in the security descriptor. **Bonus**

chatter: Note that even though Bob can remove his membership in a group, the system still knows that he’s just pretending. This is important, because the security descriptor might contain a Deny ACE for people on Project Nosebleed. Even if Bob removes the Nosebleed group membership from his token in an attempt to get around the Deny ACE, the operating system won’t be fooled: “Nice try, Bob. I know it’s still you.”

Sponsorship message: I’d like to thank my pals over in the security team for reviewing this article and making suggestions and corrections. This article is sponsored by the `AuthzAccessCheck` function, which supports caching the results of an access check.

Raymond Chen

Follow