When you call OpenThreadToken while impersonating, you have to say who is asking for the thread token

devblogs.microsoft.com/oldnewthing/20180530-00

May 30, 2018

Raymond Chen

A customer reported that `OpenThreadToken` was failing with the error `ACCESS_ DENIED` and wanted help understanding why. They shared a code fragment which operates on an account name `test` with no special privileges.

```
// Code in italics is wrong

int main()
{
  HANDLE hToken = NULL;

  // This succeeds.
  LogonUser(L"test", L".", L"test@123",
    LOGON32_LOGON_INTERACTIVE, LOGON32_PROVIDER_DEFAULT,
    &hToken);

  // This also succeeds.
  ImpersonateLoggedOnUser(hToken);

  // During this Sleep, Process Hacker shows that the thread
  // is impersonating.
  Sleep(10'000);

  // This fails with ERROR_ACCESS_DENIED.
  OpenThreadToken(GetCurrentThread(), TOKEN_QUERY,
    FALSE, &hToken);

  return 0;
}
```

According to the documentation for `OpenThreadToken`:

> *OpenAsSelf* [in]
>
> **TRUE** if the access check is to be made against the process-level security context.
>
> **FALSE** if the access check is to be made against the current security context of the thread calling the **OpenThreadToken** function.
>
> The *OpenAsSelf* parameter allows the caller of this function to open the access token of a specified thread when the caller is impersonating a token at **SecurityIdentification** level. Without this parameter, the calling thread cannot open the access token on the specified thread because it is impossible to open executive-level objects by using the **SecurityIdentification** impersonation level.

Furthermore, as I discussed some time ago,

> When a new kernel object is created, and you don't provide an explicit security descriptor for the new object, then the object is given a default security descriptor. And that default security descriptor comes from the default DACL of the token that is in effect at the point of the call.
>
> When you apply this rule to tokens, you find that, even though the behavior is consistent with other kernel objects, it also means that it is very easy to create a token that doesn't have access to itself. When you impersonate with that token, bad things happen.

The code fragment above passes `FALSE`, which means that the access check is made against the current security context, which is the impersonated test user, and that user doesn't have access to the token.

Note that changing `FALSE` to `TRUE` is only the first step in what may be a long uphill struggle. One of my colleagues on the security team added that if you don't have the **SeImpersonatePrivilege** for your process, you will run into other problems as well. The customer didn't explain the scenario where they think impersonation is a step in the solution, so it's hard to elaborate on what else can go wrong, because we don't know what they're trying to do.

Bonus reading: Changes to **SeImpersonatePrivilege** in Windows Vista.

Raymond Chen

Follow