Tyranid's Lair

Friday, 26 May 2017

Reading Your Way Around UAC (Part 2)

We left Part 1 with the knowledge that normal user processes in a split-token admin logon can get access to *Terminate*, *QueryLimitedInformation* and *Synchronize* process access rights to elevated processes. This was due to the normal user and admin user having a *Default DACL* which grants *Execute* access to the current Logon Session which is set for all tokens on the same desktop. The question we're left with is how can this possibly be used to elevate your privileges? Let's see how we can elevate our privileges prior to Windows 10.

Of the 3 access rights we have, both *Terminate* and *Synchronize* are really not that interesting. Sure you could be a dick to yourself I suppose and terminate your processes, but that doesn't seem much of interest. Instead it's *QueryLimitedInformation* which is likely to provide the most amusement, what information can we get with that access right? A quick hop, skip and jump to MSDN is in order. The following is from a page on Process Security and Access Rights:

PROCESS_QUERY_INFORMATION (0x0400)
Required to retrieve certain information about a process, such as its token, exit code, and priority class (see *OpenProcessToken*).

PROCESS_QUERY_LIMITED_INFORMATION (0x1000)
Required to retrieve certain information about a process (see *GetExitCodeProcess*, *GetPriorityClass*, *IsProcessInJob*, *QueryFullProcessImageName*). A handle that has the *PROCESS_QUERY_INFORMATION* access right is automatically granted

PROCESS_QUERY_LIMITED_INFORMATION.
Windows Server 2003 and Windows XP: This access right is not supported.

This at least confirms one thing from Part 1, that if you have *QueryInformation* access you automatically get *QueryLimitedInformation* as well. So it'd seem to make sense that *QueryLimitedInformation* just gives you a subset of what you could access from the full *QueryInformation*. And if this documentation is anything to go by all the things you could access are dull. But *QueryInformation* highlights something which *would* be very interesting to get hold of, the process token. We can double check I suppose, let's look at the documentation for OpenProcessToken to see what it says about required access.

***ProcessHandle* [in]**

A handle to the process whose access token is opened. The process must have the PROCESS_QUERY_INFORMATION access permission.

Well that seals it, nothing to see here, move along. Wait, never believe anything you read. Perhaps this is really "Fake Documentation" (*topical* if you're reading this in 2020 from a nuclear fallout shelter just ignore it). Why don't we just try it and see (make sure your previously elevated copy of *mmc.exe* is still running):

```
Use-NtObject($ps = Get-NtProcess -Name mmc.exe) {
    Get-NtToken -Primary -Process $ps[0]
} | Format-List -Property User, TokenType, GrantedAccess, IntegrityLevel
```

And then where we might expect to see an error message we instead get:

```
User           : domain\user
TokenType      : Primary
GrantedAccess  : AssignPrimary, Duplicate, Impersonate, Query,
                 QuerySource, ReadControl
IntegrityLevel : High
```

This shows we've opened the process' primary token, been granted a number of rights and to be sure we print the *IntegrityLevel* property to prove it's really a privileged token (more or less for reasons which will become clear).

What's going on? Basically the documentation is wrong, you don't need *QueryInformation* to open the process token only *QueryLimitedInformation*. You can disassemble NtOpenProcessTokenEx in the kernel if

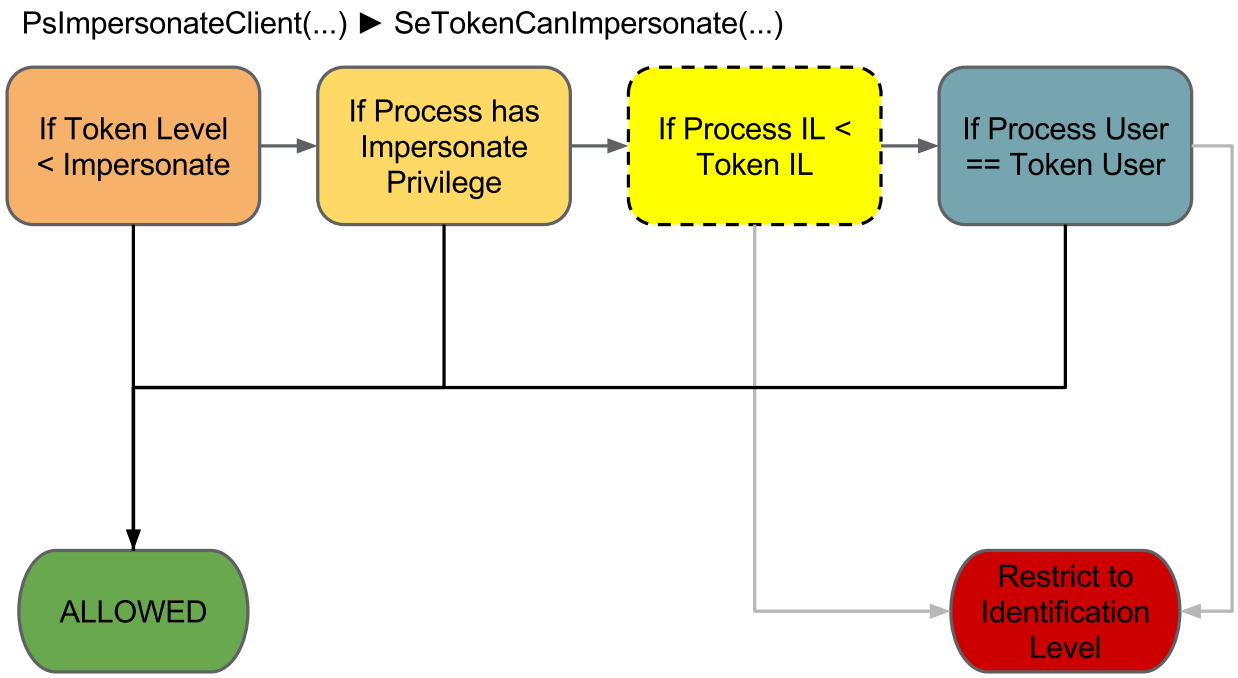
you don't believe me:

```
NTSTATUS NtOpenProcessTokenEx(HANDLE ProcessHandle,
                              ACCESS_MASK DesiredAccess,
                              DWORD HandleAttributes,
                              PHANDLE TokenHandle) {
  EPROCESS* ProcessObject;
  NTSTATUS status = ObReferenceObjectByHandle(
                        ProcessHandle,
                        PROCESS_QUERY_LIMITED_INFORMATION,
                        PsProcessType,
                        &ProcessObject,
                        NULL);
  ...
}
```

Going back to Vista it's always been the case that only *QueryLimitedInformation* was needed, contrary to the documentation. While you still need to be able to access the token through it's *DACL* it turns out that *Token* objects also use the *Default DACL* so it grants *Read* and *Execute* access to the *Logon Session SID*. But doesn't the Token have the same mandatory policy as Processes? Well let's look, we can modify the *IL Policy* dump script from Part 1 to use a token object:

```
# Get current primary token's mandatory label
$sacl = $(Get-NtToken -Primary).SecurityDescriptor.Sacl
Write-Host "Policy is $([NtApiDotNet.MandatoryLabelPolicy]$sacl[0].Mask)"
```

And the result is: "*Policy is NoWriteUp*". So while we can't modify the token (we couldn't anyway due to the *Default DACL*) we can at least read it. But again this might not seem especially interesting, what use is *Read* access? As shown earlier *Read* gives you gives you a few interesting rights, *AssignPrimary, Duplicate* and *Impersonate*. What's to stop you now creating a new Process, or Impersonating the token? Well I'd refer you to my presentation at Shakacon/Blackhat on this very topic. To cut a long story short creating a new process is virtually impossible due to the the limits imposed by the kernel function *SeIsTokenAssignableToProcess* (and the lack of the *SeAssignPrimaryTokenPrivilege*) but on the other hand impersonation takes a different approach, calling *SeTokenCanImpersonate* as shown in the following diagram.



The diagram is the rough flow chart for deciding whether a process can impersonate another token (assuming you don't have *SeImpersonatePrivilege*, which we don't). We can meet every criteria, except one. The kernel checks if the current process's *IL* is greater-or-equal to the token being impersonated. If the process *IL* is less than the token's IL then the impersonation token is dropped to *Identification* level stopping us using it to elevate our privileges. While we can't increase a token's IL we can reduce it, so all we need to do is set the token *IL* to the same as the process' IL before impersonating and in theory we should become a *Medium IL* administrator.

There is one small issue to deal with before we do that, setting the *IL* is a write operation, and we don't have write access to the token. However it turns out that as we have *Duplicate* we can call DuplicateToken which clones the entire token. We'd need get an impersonation token anyway which requires duplication so this isn't a major issue. The important fact is the resulting duplicated token gives us a *Read, Write,* and *Execute* access to the token object. As the Token object's Mandatory Label is set to the caller's *IL* (which is *Medium*) not the *IL* inside the token. This results in the kernel being able to grant us full access to the new token object, confusing I know. Note that this isn't giving us *Write* access to the original token, just a copy of it. Time for PoC||GtfO:

```
$token = Use-NtObject($ps = Get-NtProcess -Name mmc.exe) {
    Get-NtToken -Primary -Process $ps[0] -Duplicate `
    -ImpersonationLevel Impersonation `
    -TokenType Impersonation `
    -IntegrityLevel Medium
}

Use-NtObject($token.Impersonate()) {
  [System.IO.File]::WriteAllText("C:\windows\test.txt", "Hello")
}
```

And you should see it creates a text file, *C:\Windows\test.txt* with the contents *Hello*. Or you could use the New-Service cmdlet to create a new service which will run as *LocalSystem*, you're an administrator after all even if running at *Medium IL*. You might be tempted to just enable *SeDebugPrivilege* and migrate to a system process directly, but if you try that something odd happens:

```
# Will indicate SeDebugPrivilege is disabled
$token.GetPrivilege("SeDebugPrivilege").Enabled
# Try enabling the privilege.
$token.SetPrivilege("SeDebugPrivilege", $true)
# Check again, will still be disabled.
$token.GetPrivilege("SeDebugPrivilege").Enabled
```

You'll find that no matter how hard you try *SeDebugPrivilege* (and things like *SeBackupPrivilege*, *SeRestorePrivilege*) just cannot be enabled. This is another security measure that the UAC designers chose which in practice makes little realistic difference. You can't enable a small set of *GOD* privileges if the *IL* of the token is less than *High*. However you can still enable things like *SeMountVolumePrivilege* (could have some fun with that) or *SeCreateSymbolicLinkPrivilege*. We'll get back to this behavior later as it turns out to be important. Most importantly this behavior doesn't automatically disable the *Administrators* group which means we can still impersonate as a privileged user.

This works amazingly well as long as you run the example on Windows Vista, 7, 8 or 8.1. However on Windows 10 you'll get an error such as the following:

```
Use-NtObject : Exception calling "WriteAllText" with "2" argument(s): "Either a
required impersonation level was not provided, or the provided impersonation level
is invalid.
```

This error message means that the *SeTokenCanImpersonate* check check failed and the impersonation token got reverted to an *Identification* token. Clearly Microsoft know's something we don't. So that's where I'll leave it for now, come back when I post Part 3 for the conclusion, specifically getting this to work on Windows 10 and bypassing the new security checks.

Posted by tiraniddo at 06:16
Labels: Exploit, UAC, Windows

Newer Post Home Older Post