Tyranid's Lair

Thursday, 25 May 2017

Reading Your Way Around UAC (Part 1)

I'm currently in the process of trying to do some improvements to the Chrome sandbox. As part of that I'm doing updates to my Sandbox Attack Surface Analysis Toolset as I want to measure whether what I'm doing to Chrome is having a tangible security benefit. Trouble is I keep walking into UAC bypasses while I'm there which is seriously messing up the flow. So in keeping with my previous blog post on a UAC bypass let me present another. As we go I'll show some demos using the latest version of my NtObjectManager Powershell module (so think of this blog as a plug for that as well, make sure you follow the install instructions on that link before running any of the scripts).

I don't recall every seeing this issue documented (but I'm sure someone can tell me if it has been), however MS clearly know, as we'll see in a later part. Bear in mind this demonstrates just how broken UAC is in its default configuration. UAC doesn't really help you much even if you prevent the auto-elevation as this technique works as long as there exists any elevated process in the same logon session. Let this be a PSA, one of many over the years, that split-token administrator in UAC just means MS get to annoy you with prompts unnecessarily but serves very little, if not ***zero*** security benefit.

Before I start I have to address/rant about the "fileless" moniker which is bandied around for UAC bypasses. My previous blog post said it was a fileless bypass, but I still had to write to the registry (which is backed by a file) and of course some sort of executable still needs to be running (which is backed at some point by the page file) and so on. Basically all a fileless bypass means is it doesn't rely on the old *IFileOperation* tricks to hijack a DLL. Doesn't mean that at no point would some file end on disk somewhere, I suppose it's more a DFIR kind of term. Anyway enough, on to technical content.

Blog Archive

▼ 2017 (15)
- ► November (1)
- ► October (1)
- ► August (4)
- ► July (4)
- ▼ May (4)
  - Reading Your Way Around UAC (Part 3)
  - Reading Your Way Around UAC (Part 2)
  - Reading Your Way Around UAC (Part 1)
  - Exploiting Environment Variables in Scheduled Task...
- ► March (1)

► 2016 (1)
► 2015 (1)
► 2014 (9)
► 2013 (1)
► 2010 (2)

One Weird Design Decision

Oh to be a fly on the wall when Microsoft were designing UAC (or LUA as it was probably still known back then). Many different attack vectors were no doubt covered to reduce the chance of escalation from a normal user to administrator. For example Shatter Attacks (and general UI driving) was mitigated using UIPI. COM DLL planting was mitigated by making administrator processes only use HKLM for COM registrations (not especially successfully I might add). And abusing a user's resources were mitigated using Mandatory Integrity Labels to prevent write access from Low to High levels.

Perhaps there was a super secure version of UAC developed at one point, but the trouble is it would have probably been unusable. So no doubt many of the security ideas got relaxed. One of particular interest is that a non-administrator user can query some, admittedly limited, process information about administrator processes in the same desktop. This has surprising implications as we'll see.

So how much access do normal applications get? We can answer that pretty easily by running the following PS script as a normal split-token admin user.

```
Import-Module NtObjectManager

# Start mmc.exe and ensure it elevates (not really necessary for mmc)
Start-Process -Verb runas mmc.exe
Use-NtObject($ps = Get-NtProcess -Name mmc.exe) {
    $ps | Format-Table -Property ProcessId, Name, GrantedAccess
}
```

This should result in MMC elevating and the following printed to the PS console:

```
ProcessId Name                                       GrantedAccess
--------- ----                                       -------------
    17000 mmc.exe Terminate, QueryLimitedInformation, Synchronize
```

So it shows we've got 3 access rights, *Terminate*, *QueryLimitedInformation* and *Synchronize*. This kind of makes sense, after all it would be a pain if you couldn't kill processes on your desktop, or wait for them to finish, or get their name. It's at this point that the first *UAC* design decision comes into play, there exists a normal *QueryInformation* process access right, however there's a problem with using that access right, and that's down to the default *Mandatory Label Policy* (I'll refer to it just as *IL Policy* from now on) and how it's enforced on Processes.

The purpose of *IL Policy* is to specify which of the Generic Access Rights, *Read*, *Write* and *Execute* a low *IL* user can get on a resource. This is the maximum permitted access, the *IL Policy* doesn't itself grant any access rights. The user would still need to be granted the appropriate access rights in the *DACL*. So for example if the policy allows a lower *IL* process to get *Read* and *Execute*, but not *Write* (which is the default for most resources) then if the user asks for a Write access right the kernel access check will return *Access Denied* before even looking at the *DACL*. So let's look at the *IL Policy* and *Generic Access Rights* for a process object:

```
# Get current process' mandatory label
$sacl = $(Get-NtProcess -Current).SecurityDescriptor.Sacl
Write-Host "Policy is $([NtApiDotNet.MandatoryLabelPolicy]$sacl[0].Mask)"

# Get process type's GENERIC_MAPPING
$mapping = $(Get-NtType Process).GenericMapping
Write-Host "Read: $([NtApiDotNet.ProcessAccessRights]$mapping.GenericRead)"
Write-Host "Write: $([NtApiDotNet.ProcessAccessRights]$mapping.GenericWrite)"
Write-Host "Execute: $([NtApiDotNet.ProcessAccessRights]$mapping.GenericExecute)"
```

Which results in the following output:

```
Policy is NoWriteUp, NoReadUp
Read: VmRead, QueryInformation, ReadControl
Write: CreateThread, VmOperation, VmWrite, DupHandle, *Snip*
Execute: Terminate, QueryLimitedInformation, ReadControl, Synchronize
```

I've highlighted the important points. The default policy for Processes is to not allow a lower *IL* user to either *Read* or *Write*, so all they can have is *Execute* access which as we can see is what we have. However note that *QueryInformation* is a *Read* access right which would be blocked by the default IL Policy. The design decision was presumably thus, "We can't give read access, as we don't want lower IL users reading memory out of a privileged process. So let's create a new access right, *QueryLimitedInformation* which we'll assign to *Execute* and just transfer some information queries to that new right instead". Also worth noting on Vista and above you can't get *QueryInformation* access without also implicitly having *QueryLimitedInformation* so clearly MS thought enough to bodge that rather than anything else. (Thought for the reader: Why don't we get *ReadControl* access?)

Of course you still need to be able to have access in the *DACL* for those access, how come a privileged process gives these access at all? The default security of a process comes from the *Default DACL* inside the

access token which is used as the primary token for the new process, let's dump the *Default DACL* using the following script inside a normal user PS console and an elevated PS console:

```
# Get process token.
Use-NtObject($token = Get-NtToken -Primary) {
    $token.DefaultDalc |
       Format-Table @{Label="User";Expression={$_.Sid.Name}},
                    @{Label="Mask";Expression=
                         {[NtApiDotNet.GenericAccessRights]$_.Mask}}
}
```

The output as a normal user:

```
User                                                          Mask
----                                                          ----
domain\user                                                GenericAll
NT AUTHORITY\SYSTEM                                        GenericAll
NT AUTHORITY\LogonSessionId_0_295469990 GenericExecute, GenericRead
```

And again as the admin user:

```
User                                                          Mask
----                                                          ----
BUILTIN\Administrators                                     GenericAll
NT AUTHORITY\SYSTEM                                        GenericAll
NT AUTHORITY\LogonSessionId_0_295469990 GenericExecute, GenericRead
```

Once again the important points are highlighted, while the admin *DACL* doesn't allow the normal user access there is this curious *LogonSessionId* user which gets *Read* and *Execute* access. It would seem likely therefore that this must be what's giving us *Execute* access (as *Read* would be filtered by *IL Policy*). We can prove this just by dumping what groups a normal user has in their token:

```
Use-NtObject($token = Get-NtToken -Primary) {
    $token.Groups |
      Where-Object {$_.Sid.Name.Contains("LogonSessionId")} |
      Format-List
}
```

```
Name       : NT AUTHORITY\LogonSessionId_0_295469990
Sid        : S-1-5-5-0-295469990
Attributes : Mandatory, EnabledByDefault, Enabled, LogonId
```

Yup we have that group, and it's enabled. So that solves the mystery of why we get *Execute* access. This was a clear design decision on Microsoft's part to make it so a normal user could gain some level of access to an elevated process. Of course at this point you might be thinking so what? You can read some basic information from a process, how could being able to read be an issue? Well, let's see how dangerous this access is in Part 2.

Posted by tiraniddo at 17:44
Labels: Exploit, UAC, Windows

Newer Post Home Older Post