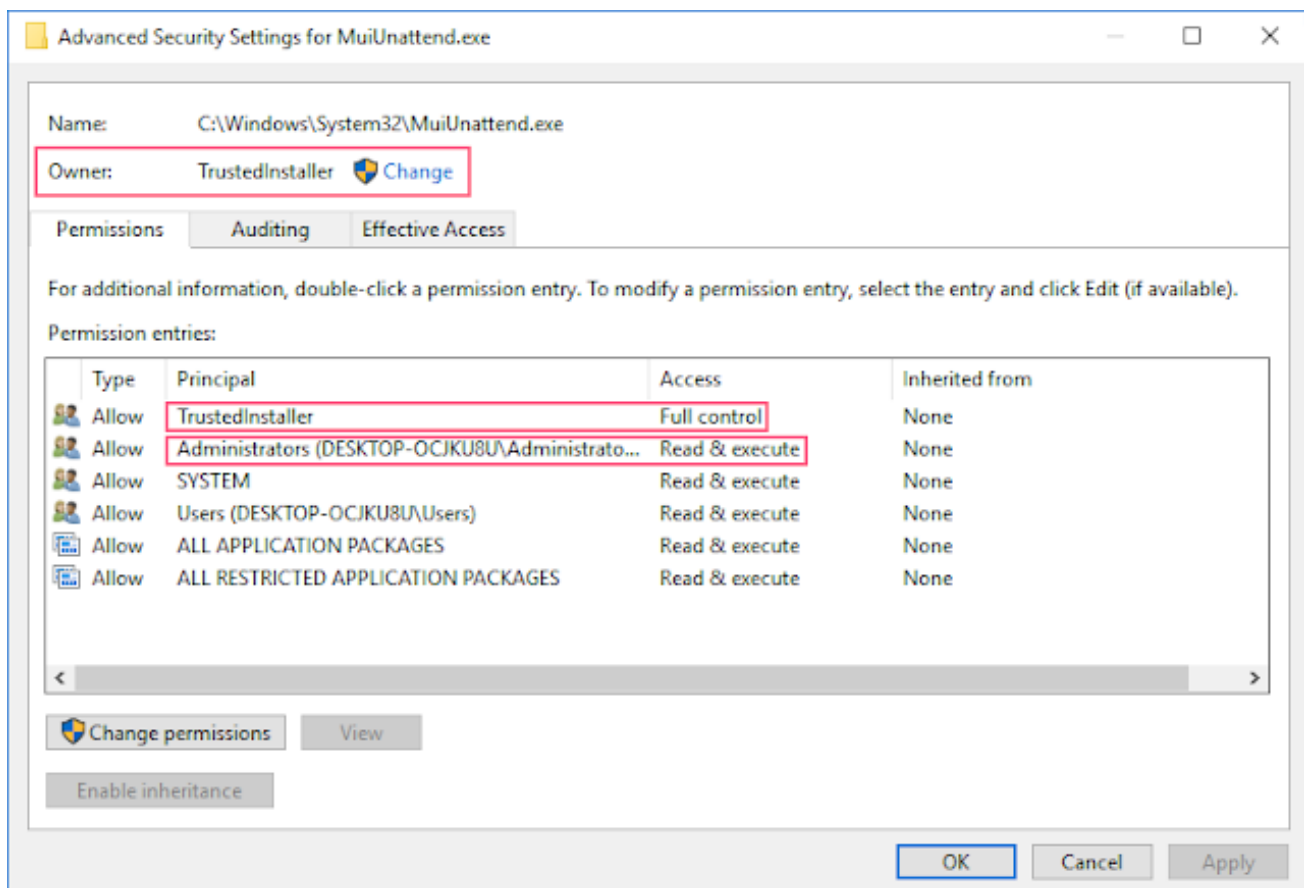


The Art of Becoming TrustedInstaller

tiraniddo.dev/2017/08/the-art-of-becoming-trustedinstaller.html

If you've spent any time administering a Windows system post Vista you'll have encountered the TrustedInstaller (TI) group which most system files and registry keys are ACL'ed to. If, for example you look at the security for a file in System32 you'll notice both that only TI can delete and modify files (not even the Administrators group is allowed) and the Owner is also TI so you can't directly change the security either.



However, if you look in the Local Users and Groups application you won't find a TI user or group. This blog post is about what the TI group really is, and more importantly, with the aid of PowerShell and the NtObjectManager module, how you can be TI to do whatever you wanted to do.

Where is TrustedInstaller?

If TI isn't a user or group then what is it? Perhaps looking at the ACL more closely will give us some insight. You can use the Get-Acl cmdlet to read the security descriptor from a file and we can list the TI ACE.

```
Select Administrator: Windows PowerShell
PS C:\> $ac1 = Get-Acl C:\Windows\system32\MuiUnattend.exe
PS C:\> $ac1.Access | Where-Object IdentityReference -match TrustedInstaller

FileSystemRights : FullControl
AccessControlType : Allow
IdentityReference : NT SERVICE\TrustedInstaller
IsInherited       : False
InheritanceFlags  : None
PropagationFlags  : None
```

We can see in the IdentityReference member that we've got the TI group, and it's prefixed with the domain "NT SERVICE". Therefore, this is a Windows Service SID. This is a feature added in Vista to allow each running service to have groups which they can use for access checks, without the overhead of adding individual real groups to the local system.

The SID itself is generated on the fly as the SHA1 hash of the uppercase version of the service name. For example the following code will calculate the actual SID:

```
$name = "TrustedInstaller"
# Calculate service SID
$bytes = [Text.Encoding]::Unicode.GetBytes($name.ToUpper())
$sha1 = [System.Security.Cryptography.SHA1]::Create()
$hash = $sha1.ComputeHash($bytes)
$rids = New-Object UInt32[] 5
[Buffer]::BlockCopy($hash, 0, $rids, 0, $hash.Length)
[string]::Format("S-1-5-80-{0}-{1}-{2}-{3}-{4}", `
    $rids[0], $rids[1], $rids[2], $rids[3], $rids[4])
```

Of course you don't need to do this, NTDLL has a RtlCreateServiceSid function, and LSASS will convert a service name to a SID and vice versa. Anyway, back to the point. What this means is that there's a service called TrustedInstaller which must be running when system resources are modified. And that's exactly what we find if we query with the SC utility.

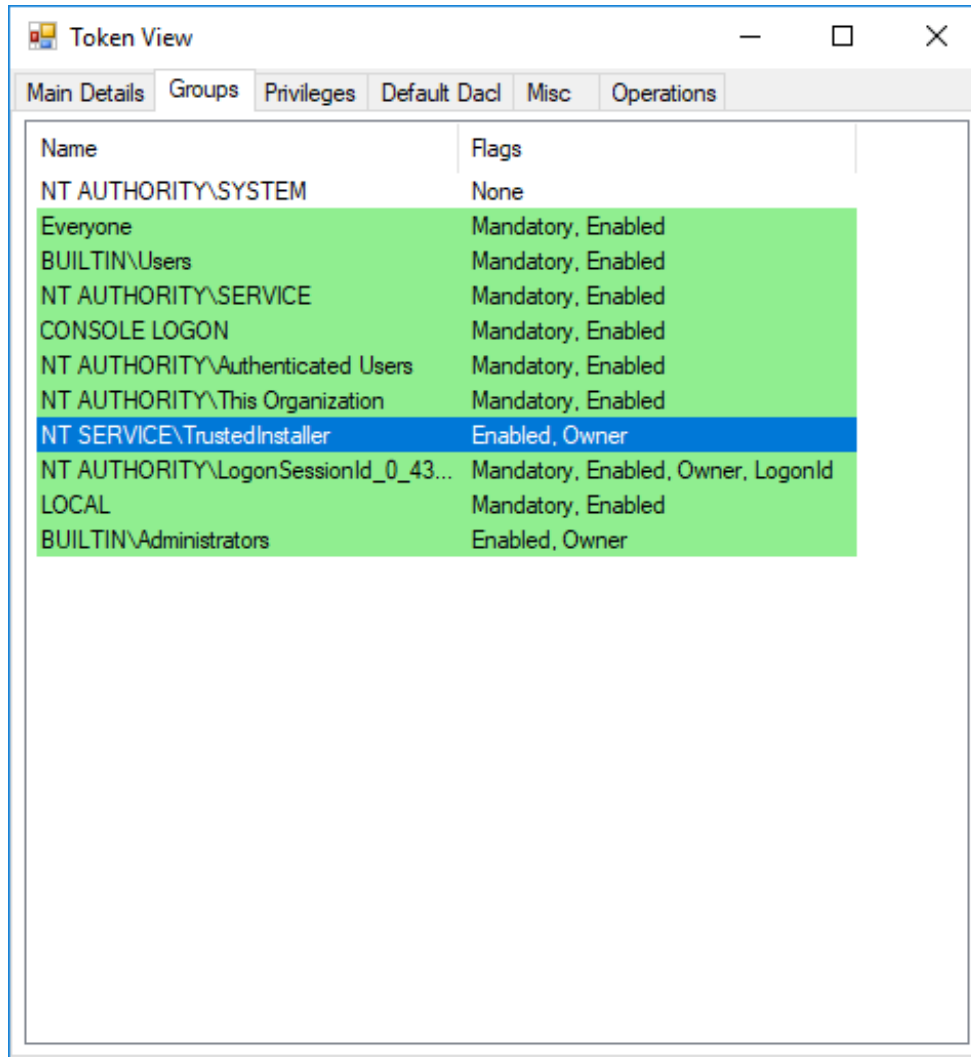
```
Select Administrator: Windows PowerShell

PS C:\> sc.exe qc TrustedInstaller
[SC] QueryServiceConfig SUCCESS

SERVICE_NAME: TrustedInstaller
        TYPE               : 10    WIN32_OWN_PROCESS
        START_TYPE          : 3    DEMAND_START
        ERROR_CONTROL       : 1    NORMAL
        BINARY_PATH_NAME    : C:\Windows\servicing\TrustedInstaller.exe
        LOAD_ORDER_GROUP    : ProfSvc_Group
        TAG                 : 0
        DISPLAY_NAME        : Windows Modules Installer
        DEPENDENCIES        :
        SERVICE_START_NAME  : LocalSystem
PS C:\> sc.exe qsidtype TrustedInstaller
[SC] QueryServiceConfig2 SUCCESS

SERVICE_NAME: TrustedInstaller
SERVICE_SID_TYPE: UNRESTRICTED
PS C:\>
```

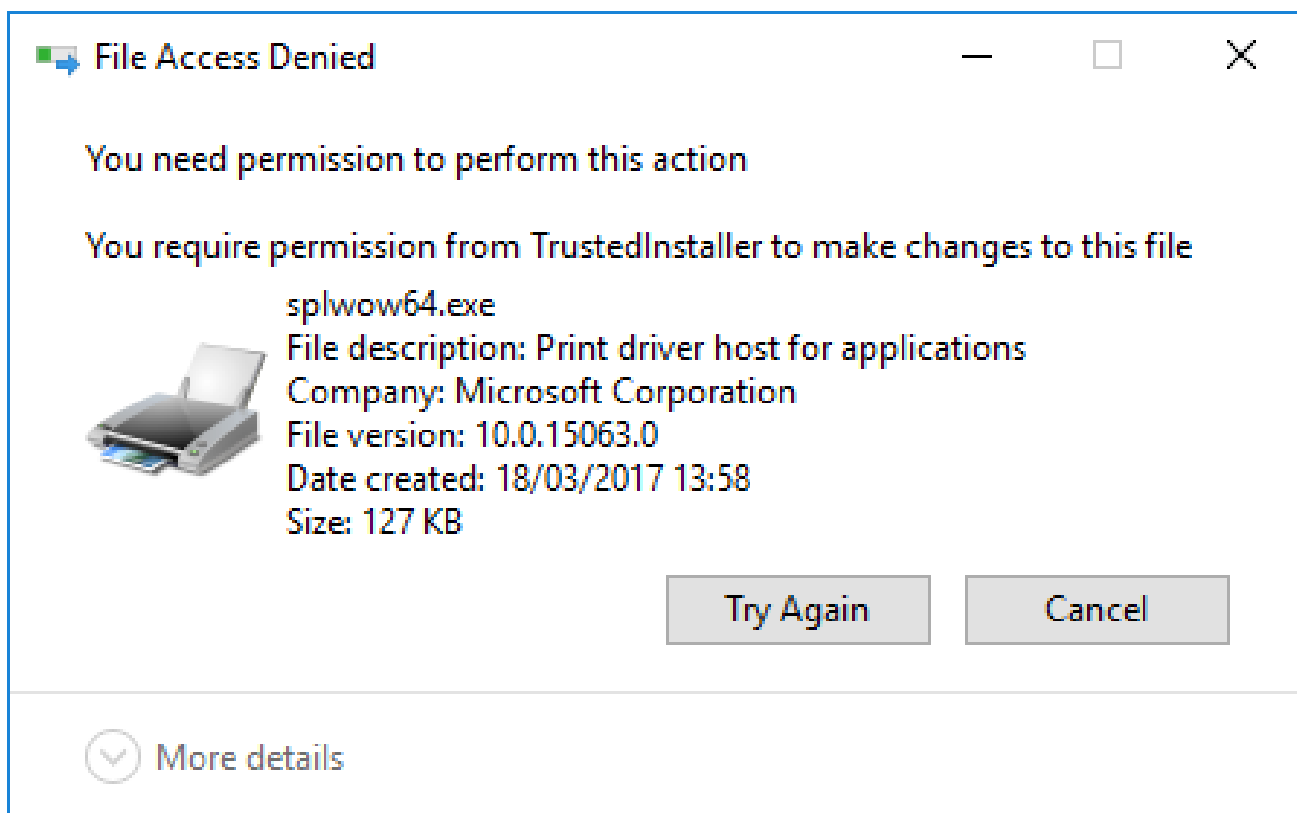
If we start the TI service and look at the Access Token we'll see that it has the TI group enabled.



Enough of the background, assuming we're an administrator how can we harness the power of TrustedInstaller?

Becoming TrustedInstaller

If you search on the Web for how to delete resources owned by TI you'll tend to find articles that advocate manually taking ownership of the file or key, then change the DACL to add the administrators group. This is because even the usually compliant IFileOperation UAC COM object won't do this automatically, as you'll end up with the following dialog.



Changing the permissions on a system file isn't exactly a great idea. If you do it wrong you could easily open up parts of your system to EoP issues, especially with directories. Explorer can make it easy to accidentally replace the security settings on all subfolders and files with little way of getting back the original values. Of course, the reason for TI is to stop you doing all this in the first place, but some people seem to really want it do it for some reason.

You might assume you could just add your current user to the TI group and that's that? Unfortunately the LSASS APIs such as `NetLocalGroupAddMembers` takes a group name not a SID, and passing "NT SERVICE\TrustedInstaller" doesn't work, as it's not a real group, but created synthetically. There might be a magic incantation to do it, or at least a low-level RPC call, but I didn't think it was worth the effort.

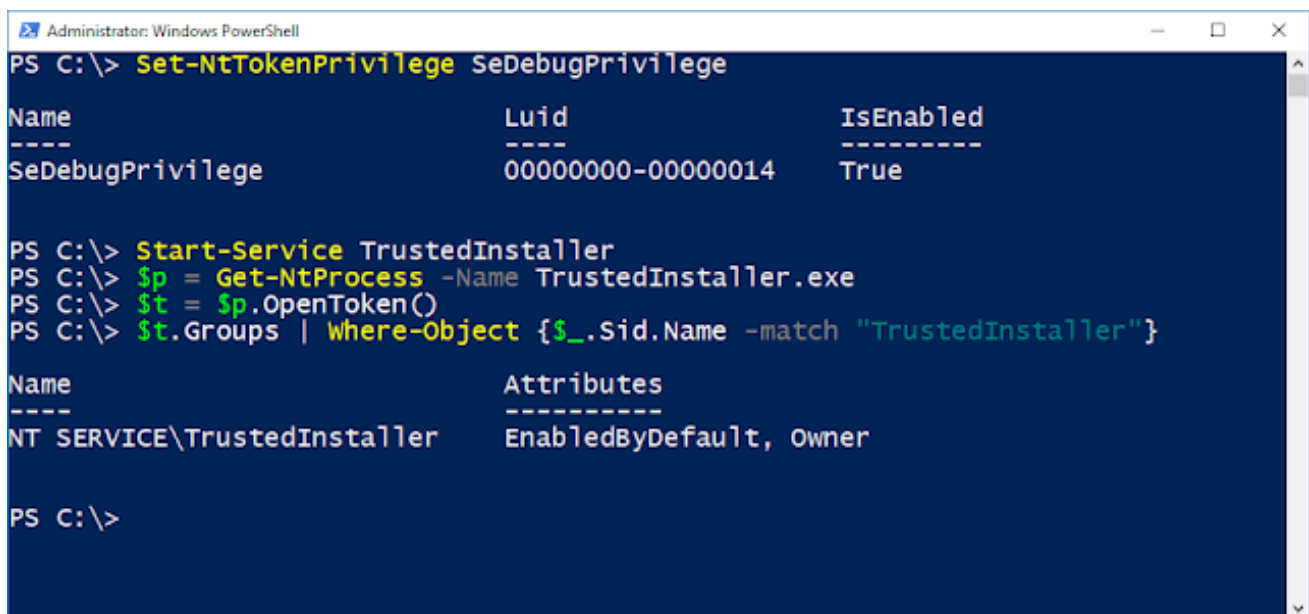
Therefore, the quick and dirty way would be to change the configuration of the TI service to run a different binary. Oddly, even though TI makes things on the system harder to mess with, it doesn't protect its own service configuration from modification by a normal administrator. So you can issue a command such as the following to delete an arbitrary file as TI:

sc config TrustedInstaller binPath= "cmd.exe /C del path\tofile"

Start the TI service and *bang* the file is gone. Another reason this works is TI is not a Protected Process Light (PPL), which is odd because the TI group is given special permission to stop and delete PPL services. I pointed [this out](#) to MSRC (and Alex Ionescu did so in 2013, and clearly I didn't bother to read it) but they didn't do anything to fix it as no matter what they pretend PPL isn't a security boundary, well until it a security boundary.

This still feels like a hack, you'd have to restore the TI service to its original state otherwise things like Windows Update will get unhappy really quickly. As the TI service has a token with the correct groups, what about just starting the service then "borrowing" the Token from it to create a new process or for impersonation?

As an admin we've got SeDebugPrivilege so can just open the TI process and open it's token. Then we can do whatever we like with it. Simple really, let's give that a try.



```
Administrator: Windows PowerShell
PS C:\> Set-NtTokenPrivilege SeDebugPrivilege

Name                Luid                IsEnabled
----                -
SeDebugPrivilege    00000000-00000014  True

PS C:\> Start-Service TrustedInstaller
PS C:\> $p = Get-NtProcess -Name TrustedInstaller.exe
PS C:\> $t = $p.OpenToken()
PS C:\> $t.Groups | Where-Object {$_.Sid.Name -match "TrustedInstaller"}

Name                Attributes
----                -
NT SERVICE\TrustedInstaller  EnabledByDefault, Owner

PS C:\>
```

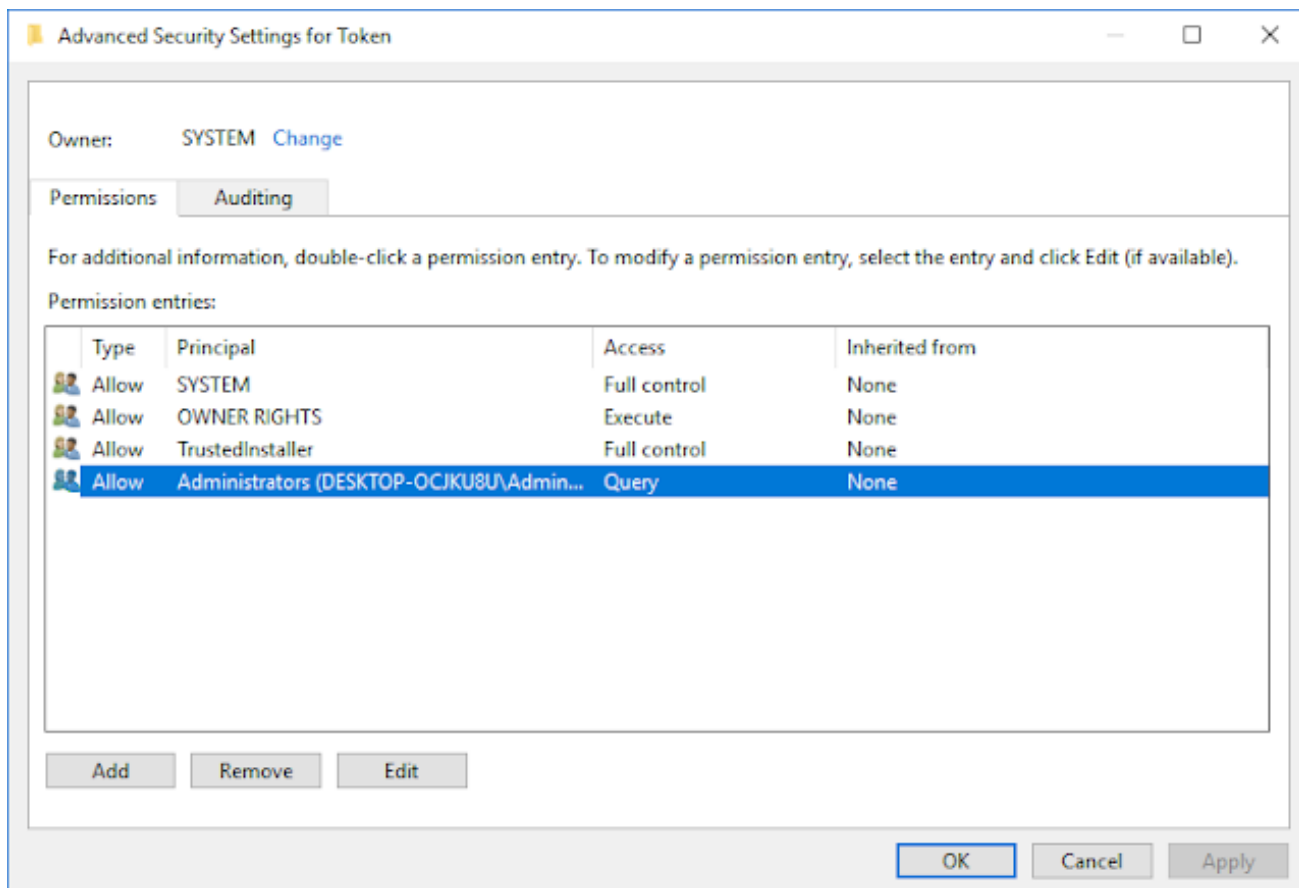
Well that was easy. Pat yourself on the back, grab a cold one it's time for some Trusted Installing...

```
Administrator: Windows PowerShell
PS C:\> $proc = New-Win32Process cmd.exe -CreationFlags NewConsole -Token $t
Exception calling "CreateProcessAsUser" with "2" argument(s): "Access is
denied"
At C:\Users\user\Documents\WindowsPowerShell\Modules\NtObjectManager\1.0.9\Ntob
jectManager.psm1:515 char:3
+ [SandboxAnalysisUtils.Win32Process]::CreateProcessAsUser($Tok ...
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [], MethodInvocationException
+ FullyQualifiedErrorId : Safewin32Exception

PS C:\> $imp = $t.Impersonate()
Exception calling "Impersonate" with "0" argument(s): "(0xC0000022) - {Access
Denied}
A process has requested access to an object, but has not been granted those
access rights."
At line:1 char:1
+ $imp = $t.Impersonate()
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [], MethodInvocationException
+ FullyQualifiedErrorId : NtException

PS C:\> $t.GrantedAccess
Query
PS C:\>
```

Well crap, seems we can't create a new process or impersonate the token. That's not much good. At the bottom of the screenshot we can see why, the token we've got only has TOKEN_QUERY access. We typically need at least TOKEN_DUPLICATE access to get the primary token for a new process or to create an impersonation token. Checking the security descriptor of the token using Process Hacker (we don't even have READ_CONTROL to read it from PS) explains the reason why we've got such limited access.



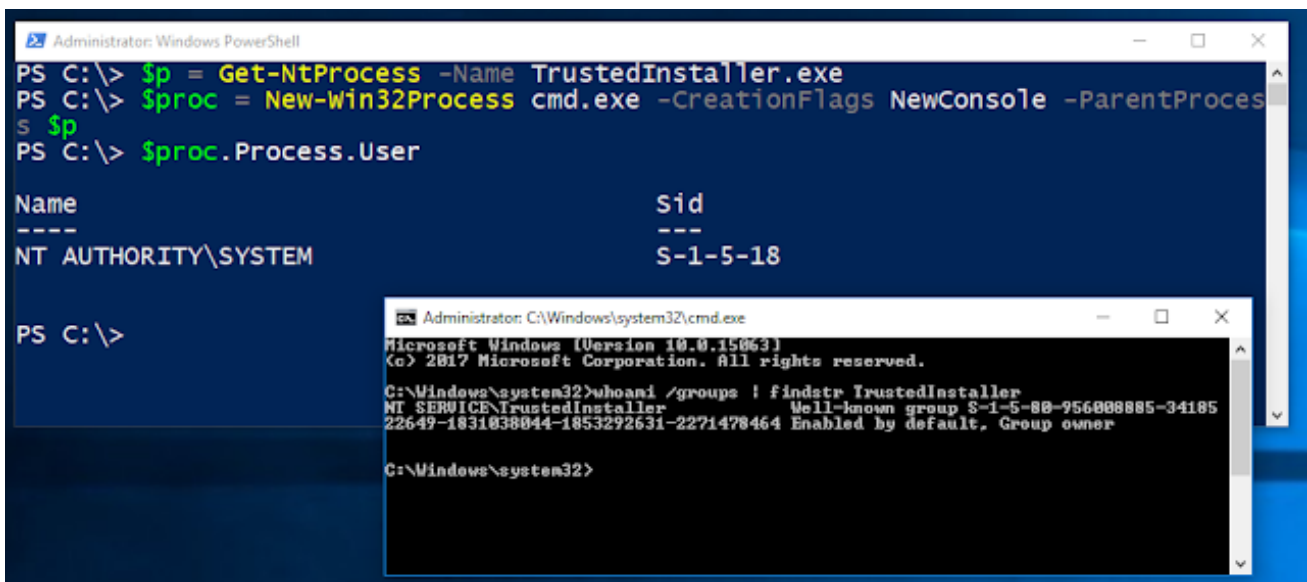
We can see that the Administrators group only has TOKEN_QUERY access which at least matches up with the access we were granted on the token object. You might wonder why SeDebugPrivilege didn't help here. The Debug privilege only bypass the security checks on Process and Thread objects, it doesn't do anything on Tokens so we get no help. Are we stuck now, at least without the more destructive techniques such as changing the service binary?

Of course not. There are examples of how to get stuff running as TI such as [this](#) but they seem to rely on first getting code running at system by install a service (like psexec with the -s switch) and then stealing the the TI token and creating a new process. Needless to say, if I wanted to create a service I'd just modify the TrustedInstaller service to begin with :-)

So here's two quick tricks to allow use to get around this permission limitation which doesn't require any new or modified services or injecting shellcode. First, let's deal with creating a new process. The parent of a new process is the caller of CreateProcess, however for UAC this would make all elevated processes children of the UAC service which would look

somewhat odd. To support the principle of least surprise MS introduced in Vista the ability to specify an explicit parent process when creating a new process so that the elevated process could still be a child of the caller.

Normally in the UAC case you specify an explicit token to assign to the new process. However, if you don't specify a token the new process will inherit from the assigned parent, the only requirement for this is the handle to the process we use as a parent must have the PROCESS_CREATE_PROCESS access right. As we've got SeDebugPrivilege we can get full access to the TI process including the right to create new child processes from it. As an added bonus the kernel process creation code will even assign the correct session ID for the caller so we can create interactive processes. We exploit this behavior to create an arbitrary process running as the TI service token on the current desktop using the New-Win32Process cmdlet and passing the process object in the -ParentProcess parameter.



```
Administrator: Windows PowerShell
PS C:\> $p = Get-NtProcess -Name TrustedInstaller.exe
PS C:\> $proc = New-Win32Process cmd.exe -CreationFlags NewConsole -ParentProcess $p
PS C:\> $proc.Process.User

Name                               Sid
----                               -
NT AUTHORITY\SYSTEM                S-1-5-18

PS C:\>
```

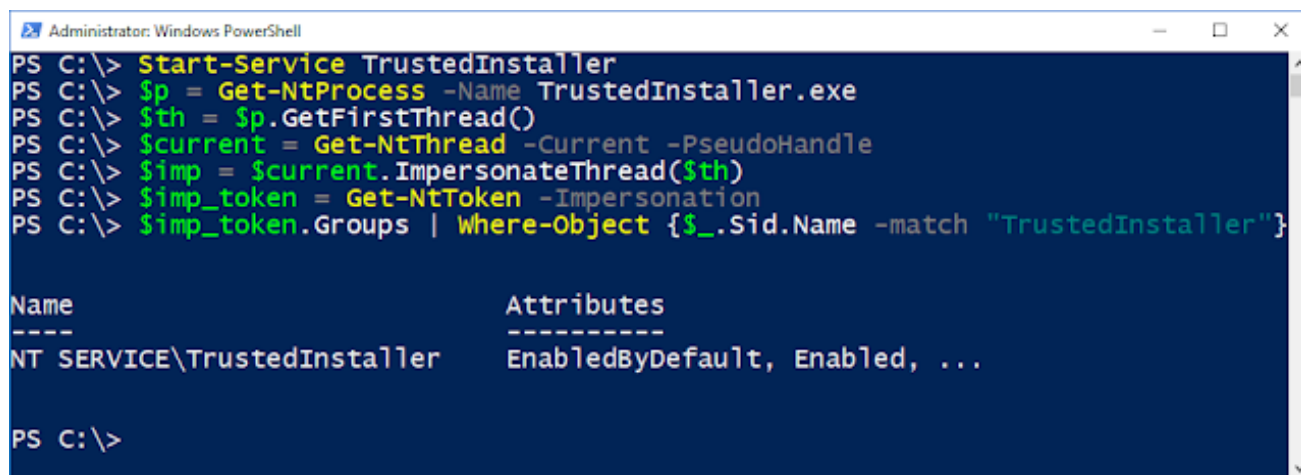
```
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Windows\system32>whoami /groups | findstr TrustedInstaller
NT SERVICE\TrustedInstaller Well-known group S-1-5-80-956888885-34185
22649-1831838844-1853292631-2271478464 Enabled by default, Group owner

C:\Windows\system32>
```

It'd be kind of useful to impersonate the token as well without creating a new process. Is there a way we can achieve that? Sure, we can use the NtImpersonateThread API, which allows you capture the impersonation context from an existing thread and apply it to another. The way impersonation contexts work is the kernel will try and capture the impersonation token for the thread first. If there is no existing impersonation token then it'll take a copy of the primary token of the process associated with thread and impersonate that instead. And the beauty of the NtImpersonateThread API, like setting the parent process, doesn't require permission to access the token, it just required THREAD_DIRECT_IMPERSONATION access to a thread which we can get due to SeDebugPrivilege. We can therefore get an impersonation thread without a new process by the following steps:

1. Open process with at least PROCESS_QUERY_INFORMATION access to list its threads.
2. Open the first thread in the process with THREAD_DIRECT_IMPERSONATION access. We'll assume that the thread isn't impersonating some lowly user.
3. Call NtImpersonateThread to steal an impersonation token.



```
PS C:\> Start-Service TrustedInstaller
PS C:\> $p = Get-NtProcess -Name TrustedInstaller.exe
PS C:\> $th = $p.GetFirstThread()
PS C:\> $current = Get-NtThread -Current -PseudoHandle
PS C:\> $imp = $current.ImpersonateThread($th)
PS C:\> $imp_token = Get-NtToken -Impersonation
PS C:\> $imp_token.Groups | Where-Object {$_.Sid.Name -match "TrustedInstaller"}
```

Name	Attributes
NT SERVICE\TrustedInstaller	EnabledByDefault, Enabled, ...

```
PS C:\>
```

Now as long as something else doesn't set the main thread's impersonation token (and you don't do anything on a separate thread) then your PS console we act as if it has the TI group enabled. Handy :-)

Wrap-Up

Hopefully this has given you some information about what TrustedInstaller and a few tricks to get hold of a token for that group using an admin account which would not normally be allowed. This applies equally to a number of different system services on modern Windows, which can be pain if you want to interact with their resources for testing purposes such as using my Sandbox Tools to work out what resources they can access.

Update (20170821)

Thought I should at least repeat that of course there's many ways of getting the TI token other than these 3 techniques. For example as Vincent Yiu pointed out on [Twitter](#) if you've got easy access to a system token, say using Metasploit's *getsystem* command you can impersonate system and then open the TI token, it's just IMO less easy :-). If you get a system token with *SeTcbPrivilege* you can also call [LogonUserExExW](#) or [LsaLogonUser](#) where you can specify an set of additional groups to apply to a service token. Finally if you get a system token with *SeCreateTokenPrivilege* (say from LSASS.exe if it's not running PPL) you can craft an arbitrary token using the *NtCreateToken* system call.

