Sudo On Windows a Quick Rundown

tiraniddo.dev/2024/02/sudo-on-windows-quick-rundown.html

Background

The Windows Insider Preview build 26052 just shipped with a sudo command, I thought I'd just take a quick peek to see what it does and how it does it. This is only a short write up of my findings, I think this code is probably still in early stages so I wouldn't want it to be treated too harshly. You can see the official announcement here.

To run a command using sudo you can just type:

C:\> sudo powershell.exe

The first thing to note, if you know anything about the security model of Windows (maybe buy my book, hint hint), is that there's no equivalent to SUID binaries. The only way to run a process with a higher privilege level is to get an existing higher privileged process to start it for you or you have sufficient permissions yourself though say SeImpersonatePrivilege or SeAssignPrimaryToken privilege and have an access token for a more privileged user. Since Vista, the main way of facilitating running more privileged code as a normal user is to use UAC. Therefore this is how sudo is doing it under the hood, it’s just spawning a process via UAC using the ShellExecute runas verb.

This is slightly disappointing as I was hoping the developers would have implemented a sudo service running at a higher privilege level to mediate access. Instead this is really just a fancy executable that you can elevate using the existing UAC mechanisms.

The other sad thing is, as is Microsoft tradition, this is a sudo command in name only. It doesn’t support any policies which would allow a user to run specific commands elevated, either with a password requirement or without. It’ll just run anything you give it, and only if that user can pass a UAC elevation prompt.

There are four modes of operation that can be configured in system settings, why this needs to be a system setting I don’t really know.

Initially sudo is disabled, running the sudo command just prints “Sudo is disabled on this machine. To enable it, go to the Developer Settings page in the Settings app”. This isn’t because of some fundamental limit on the behavior of the sudo implementation, instead it’s just an Enabled value in HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Sudo which is set to 0.

The next option (value 1) is to run the command in a new window. All this does is pass the command line you gave to sudo to ShellExecute with the runas verb. Therefore you just get the normal UAC dialog showing for that command. Considering the general move to using PowerShell for everything you can already do this easily enough with the command:

```
PS> Start-Process -Verb runas powershell.exe
```

The third and fourth options (value 2 and 3) are “With input disabled” and “Inline”. They’re more or less the same, they can run the command and attach it to the current console window by sharing the standard handles across to the new process. They use the same implementation behind the scenes to do this, a copy of the sudo binary is elevated with the command line and the calling PID of the non-elevated sudo. E.g. it might try and running the following command via UAC:

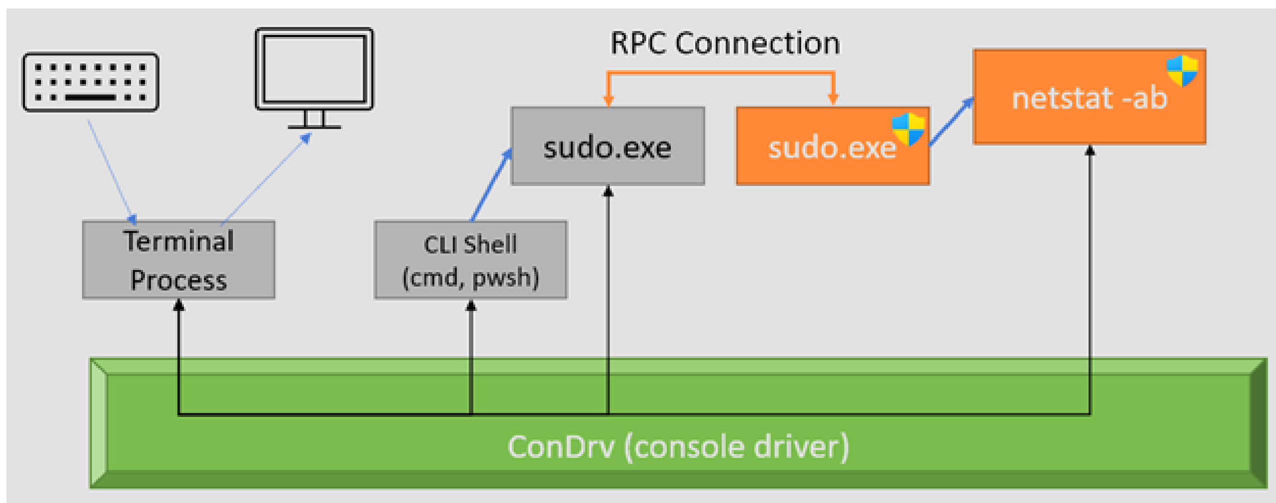
```
C:\> sudo elevate -p 1234 powershell.exe
```

Oddly, as we’ll see passing the PID and the command seems to be mostly unnecessary. At best it’s useful if you want to show more information about the command in the UAC dialog, but again as we’ll see this isn’t that useful.

The only difference between the two is “With input disabled” you can only output text from the elevated application, you can’t interact with it. Whereas the Inline mode allows you to run the command elevated in the same console session. This final mode has the obvious risk that the command is running elevated but attached to a low privileged window. Malicious code could inject keystrokes into that console window to control the privileged process. This was pointed out in the Microsoft blog post linked earlier. However, the blog does say that running it with input disabled mitigates this issue somewhat, as we’ll see it does not.

How It Really Works

For the “New Window” mode all sudo is doing is acting as a wrapper to call ShellExecute. For the inline modes it requires a bit more work. Again go back and read the Microsoft blog post, tbh it gives a reasonable overview of how it works. In the blog it has the following diagram, which I’ll reproduce here in case the link dies.



What always gets me interested is where there's an RPC channel involved. The reason a communications channel exists is due to the limitations of UAC, it very intentionally doesn't allow you to attach elevated console processes to an existing low privileged console (grumble UAC is not a security boundary, but then why did this do this if it wasn't grumble). It also doesn't pass along a few important settings such as the current directory or the environment which would be useful features to have in a sudo like command. Therefore to do all that it makes sense for the normal privileged sudo to pass that information to the elevated version.

Let's check out the RPC server using NtObjectManager:

```
PS> $rpc = Get-RpcServer C:\windows\system32\sudo.exe

PS> Format-RpcServer $rpc

[

  uuid(F691B703-F681-47DC-AFCD-034B2FAAB911),

  version(1.0)

]

interface intf_f691b703_f681_47dc_afcd_034b2faab911 {

    int server_PrepareFileHandle([in] handle_t _hProcHandle, [in] int p0, [in, system_handle(sh_file)]
HANDLE p1);

    int server_PreparePipeHandle([in] handle_t _hProcHandle, [in] int p0, [in, system_handle(sh_pipe)]
HANDLE p1);
```

```
    int server_DoElevationRequest([in] handle_t _hProcHandle, [in, system_handle(sh_process)] HANDLE p0, [in] int p1, [in, string] char* p2, [in, size_is(p4)] byte* p3[], [in] int p4, [in, string] char* p5, [in] int p6, [in] int p7, [in, size_is(p9)] byte* p8[], [in] int p9);

    void server_Shutdown([in] handle_t _hProcHandle);

}
```

Of the four functions, the key one is server_DoElevationRequest. This is what actually does the elevation. Doing a quick bit of analysis it seems the parameters correspond to the following:

HANDLE p0 - Handle to the calling process.

int p1 - The type of the new process, 2 being input disabled, 3 being inline.

char* p2 - The command line to execute (oddly, in ANSI characters)

byte* p3[] - Not sure.

int p4 - Size of p3.

char* p5 - The current directory.

int p6 - Not sure, seems to be set to 1 when called.

int p7 - Not sure, seems to be set to 0 when called.

byte* p8 - Pointer to the environment block to use.

int p9 - Length of environment block.

The RPC server is registered to use ncalrpc with the port name being sudo_elevate_PID where PID is just the value passed on the elevation command line for the -p argument. The PID isn't used for determining the console to attach to, this is instead passed through the HANDLE parameter, and is only used to query its PID to pass to the AttachConsole API.

Also as said before as far as I can tell the command line you want to execute which is also passed to the elevated sudo is unused, it's in fact this RPC call which is responsible for executing the command properly. This results in something interesting. The elevated copy of sudo doesn't exit once the new process has

started, it in fact keeps the RPC server open and will accept other requests for new processes to attach to. For example you can do the following to get a running elevated sudo instance to attach an elevated command prompt to the current PowerShell console:

```
PS> $c = Get-RpcClient $rpc

PS> Connect-RpcClient $c -EndpointPath sudo_elevate_4652

PS> $c.server_DoElevationRequest((Get-NtProcess -ProcessId $pid), 3, "cmd.exe", @(), 0, "C:\", 1, 0, @(), 0)
```

There are no checks for the caller's PID to make sure it's really the non-elevated sudo making the request. As long as the RPC server is running you can make the call. Finding the ALPC port is easy enough, you can just enumerate all the ALPC ports in \RPC Control to find them.

A further interesting thing to note is that the type parameter (p1) doesn't have to match the configured sudo mode in settings. Passing 2 to the parameter runs the command with input disabled, but passing any other value runs in the inline mode. Therefore even if sudo is configured in new window mode, there's nothing stopping you running the elevated sudo manually, with a trusted Microsoft signed binary UAC prompt and then attaching the inline mode via the RPC service. E.g. you can run sudo using the following PowerShell:

```
PS> Start-Process -Verb runas -FilePath sudo -ArgumentList "elevate", "-p", 1111, "cmd.exe"
```

Fortunately sudo will exit immediately if it's configured in disabled mode, so as long as you don't change the defaults it's fine I guess.

I find it odd that Microsoft would rely on UAC when UAC is supposed to be going away. Even more so that this command could have just been a PowerToy as other than the settings UI changes it really doesn't need any integration with the OS to function. And in fact I'd argue that it doesn't need those settings either. At any rate, this is no more a security risk than UAC already is, or is it…

Looking back at how the RPC server is registered can be enlightening:

```
RPC_STATUS StartRpcServer(RPC_CSTR Endpoint) {
```

```
  RPC_STATUS result;

  result = RpcServerUseProtseqEpA("ncalrpc",
      RPC_C_PROTSEQ_MAX_REQS_DEFAULT, Endpoint, NULL);
  if ( !result )
  {
    result = RpcServerRegisterIf(server_sudo_rpc_ServerIfHandle, NULL, NULL);
    if ( !result )
      return RpcServerListen(1, RPC_C_PROTSEQ_MAX_REQS_DEFAULT, 0);
  }
  return result;
}
```

Oh no, that’s not good. The code doesn’t provide a security descriptor for the ALPC port and it calls RpcServerRegisterIf to register the server, which should basically never be used. This old function doesn’t allow you to specify a security descriptor or a security callback. What this means is that any user on the same system can connect to this service and execute sudo commands. We can double check using some PowerShell:

```
PS> $as = Get-NtAlpcServer
PS> $sudo = $as | ? Name -Match sudo
PS> $sudo.Name
sudo_elevate_4652
PS> Format-NtSecurityDescriptor $sudo -Summary
<Owner> : BUILTIN\Administrators
<Group> : DESKTOP-9CF6144\None
<DACL>
```

Everyone: (Allowed)(None)(Connect|Delete|ReadControl)

NT AUTHORITY\RESTRICTED: (Allowed)(None)(Connect|Delete|ReadControl)

BUILTIN\Administrators: (Allowed)(None)(Full Access)

BUILTIN\Administrators: (Allowed)(None)(Full Access)

Yup, the DACL for the ALPC port has the Everyone group. It would even allow restricted tokens with the RESTRICTED SID set such as the Chromium GPU processes to access the server. This is pretty poor security engineering and you wonder how this got approved to ship in such a prominent form.

The worst case scenario is if an admin uses this command on a shared server, such as a terminal server then any other user on the system could get their administrator access. Oh well, such is life…

I will give Microsoft props though for writing the code in Rust, at least most of it. Of course it turns out that the likelihood that it would have had any useful memory corruption flaws to be low even if they'd written it in ANSI C. This is a good lesson on why just writing in Rust isn't going to save you if you end up just introducing logical bugs instead.