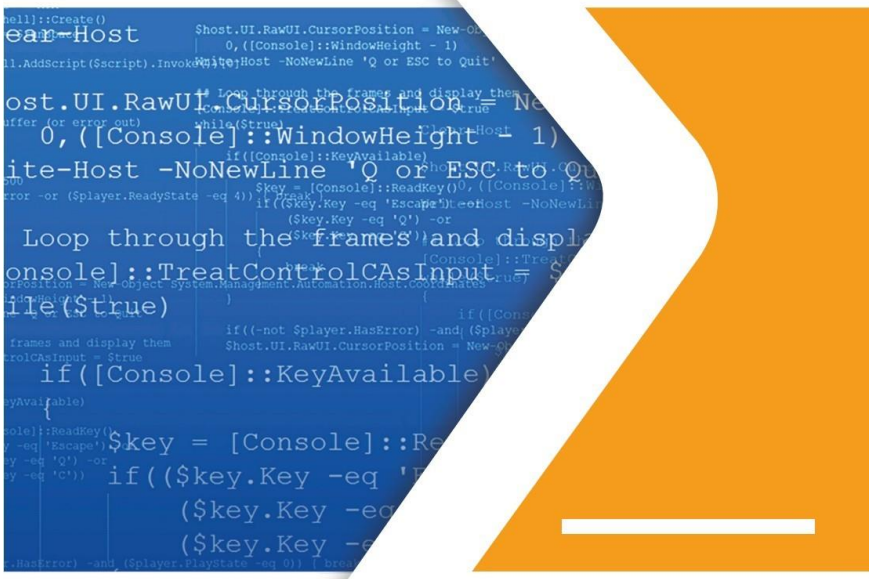
ScriptRunner®

PowerShell Security

SCRIPTRUNNER EDITION



Michael Pietroforte
Wolfgang Sommergut

Windows Pro
Windows, Virtualisierung und Cloud für Profis

PowerShell Security

- Limit language features
- Secure communication
- Track abuse

Michael Pietroforte
Wolfgang Sommergut

Cover Designer: Claudia Wolff

1. Edition 2020
ISBN: 9781672847827

Michael Pietroforte
Wolfgang Sommergut

PowerShell Security

Limit language features, secure communications, track abuse

- Control execution of scripts using execution policy, code signing and constrained language mode
- Secure PowerShell remoting with SSH und TLS
- Delegate administrative tasks with JEA
- Audit and analyze PowerShell activities, encrypt logs
- Improve code quality following best practices

About the authors

Michael Pietroforte is the founder and editor in chief of 4sysops. He has more than 35 years of experience in IT management and system administration.

Wolfgang Sommergut has over 20 years of experience in IT journalism. He has also worked as a system administrator and as a tech consultant.

Today he runs the German publication WindowsPro.de.

Table of contents

1 PowerShell as a hacking tool: Prevent abuse of scripts

PowerShell is a powerful tool for system administration and as such also a perfect means for hackers. Due to the tight integration into the system, attempts to simply block PowerShell provide a false impression of security. The best protection is provided by PowerShell's own mechanisms.

PowerShell offers almost unlimited access to the resources of a Windows computer and also can automate numerous applications such as Exchange. Users aren't limited to the many modules and cmdlets, but can also integrate .NET classes, Windows APIs, and COM objects. These capabilities are particularly dangerous in the hands of attackers.

Since many versions of With Windows Server, Microsoft avoids to activate any roles and features on a freshly installed machine in order to minimize the attack surface. On such a locked down system users must explicitly add all required services.

1.1 Lax default configuration of PowerShell

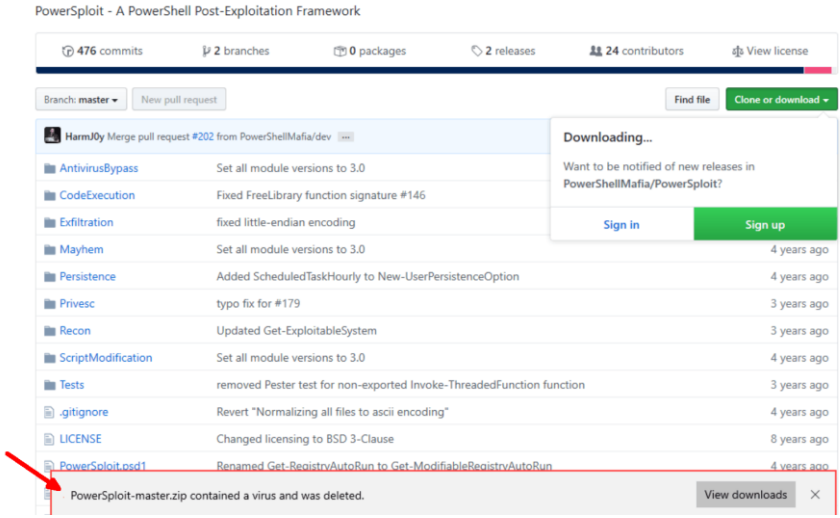
However with PowerShell, the full range of functions is available from the start on every Windows PC, if you put aside the "protection" by a restrictive execution policy. However, it is not recommended to leave this state as it is.

You don't only have to fear malicious PowerShell experts who can exploit all potentials of a script. In fact, even basic knowledge is sufficient to penetrate systems with the help of various hacking tools.

1.2 Hacking tools for PowerShell

Quite a number of them can be easily obtained as open source via Github. These include the extensive script and module collections PowerSploit, PowerShell Empire, Nishang or PowerUp.

You might assume that your computers are well protected by virus scanners which detect and block these hacking tools. In fact, Windows Defender, for example, intervenes after the download and quarantines the scripts.



Windows Defender prevents the download of PowerSploit

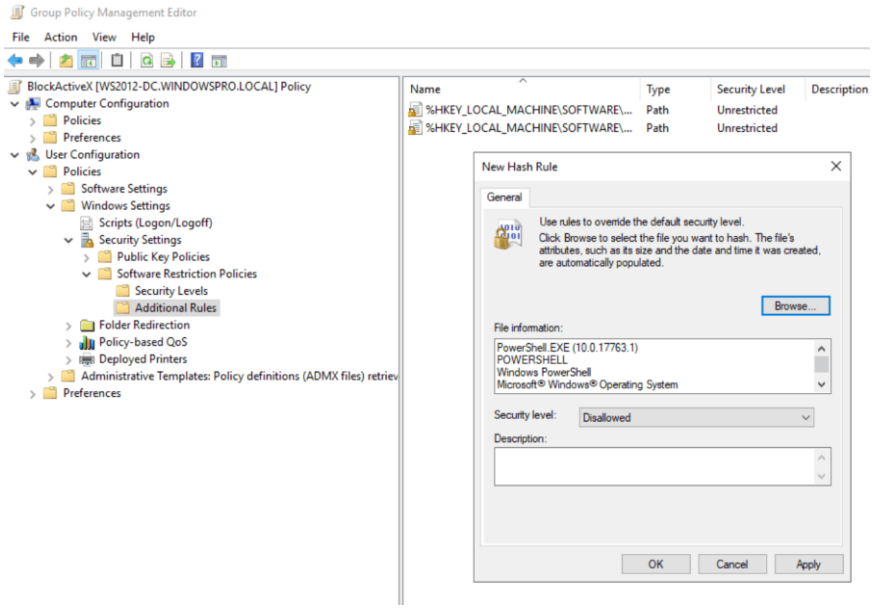
However, in contrast to binary files, scripts can be changed quite easily to fool a signature based recognition. For example, you can copy Invoke-Mimikatz from the browser window and paste it into an editor like PowerShell_ISE to experiment with the code.

This blog post by Carrie Roberts demonstrates how to outwit most virus scanners by searching and replacing a few significant code snippets. At this point, the technique discussed there may not be up to date any more, but a bit of experimenting will probably reveal how virus scanners detect this script. Otherwise, various AMSI-Bypasses can help you to overwhelm Windows Defender.

1.3 General blocking of PowerShell

To prevent such threats, many companies will take a radical measure and disable PowerShell altogether. In centrally managed environments, black-listing with AppLocker or the Software Restriction Policies is an effective solution.

If you decide to use the software restriction, you create two new hash rules and connect them to *powershell.exe* and *powershell_ise.exe*. For the security level choose *Not allowed*. If you block the programs at the user level, admins can be excluded.



Blocking powershell.exe with software restriction policies

This approach has two disadvantages. Firstly, it can be an obstacle to system administration, because PowerShell has become an indispensable

tool for most admins. For example, PowerShell logon scripts that are executed in the security context of a user will no longer work.

1.4 Circumvention through alternative shells

More serious, however, is that PowerShell comprises more than just powershell.exe or power-shell_ise.exe and therefore cannot be permanently blocked by denying access to these two files. Rather, it is a system component (System.Management.Automation) that cannot be removed and can be used by various runspaces.

Attackers could thus access PowerShell from any of their own programs. It is therefore no surprise that already shells exist that can be integrated into your own code or that can be executed directly. Among them are p0wnedShell or PowerOPS.

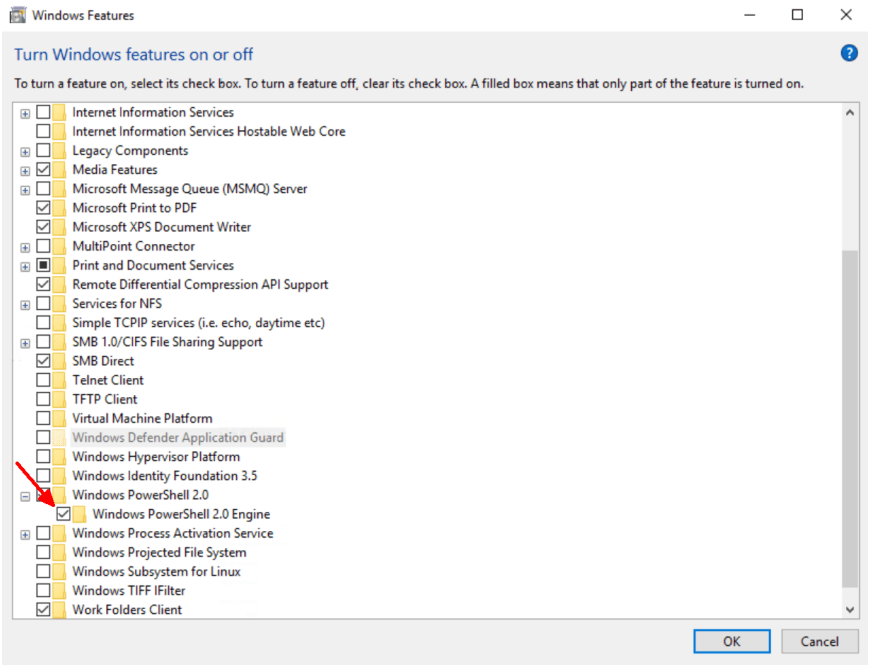
In addition, numerous versions of PowerShell 6 and 7 are available for download in ZIP format, which can be easily unpacked into a directory and executed. Frequent previews of PowerShell 7 would keep admins busy, because they always have to create new rules to cover all these versions.

And last but not least, another workaround is to compile PowerShell-Scripts into executable files. They are also not dependent on powershell.exe.

1.5 Secure PowerShell with integrated mechanisms

Instead of completely banishing PowerShell without achieving real security, it makes more sense to use its security features. These were further improved with version 5, so that you should update PCs to the latest version of PowerShell.

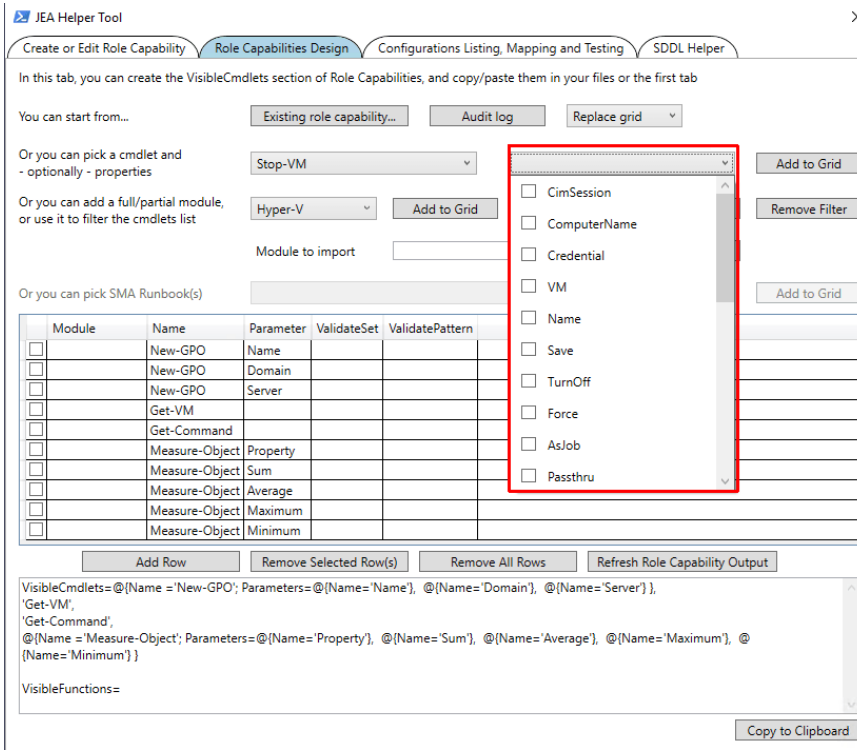
It is also highly recommended to remove PowerShell 2.0, which is still preinstalled as an optional feature and can be uninstalled in Windows 8.1 and Server 2012 or higher. With this old version, all major restrictions for PowerShell can be circumvented.



PowerShell 2.0 is an optional feature starting with Windows 8 and Server 2012 and is enabled by default.

One of the key security mechanisms of Windows PowerShell is the Constrained Language Mode, which disables several dangerous features. This language mode is particularly effective when used in conjunction with application whitelisting.

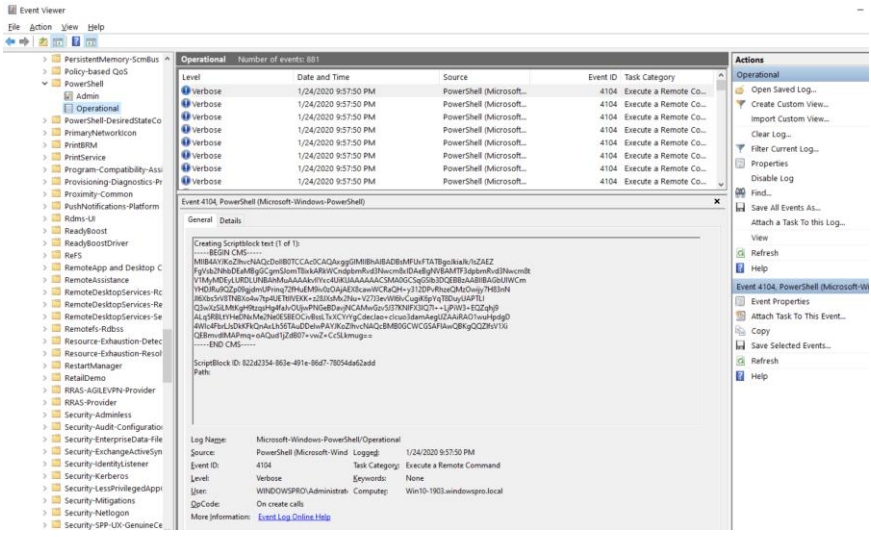
When running PowerShell on remote machines Session Configurations and Just Enough Administration can effectively limit the scope for users.



Selecting the allowed parameters of a cmdlet for JEA

Besides the means to prevent the abuse of PowerShell, there are also functions to track down suspicious and unwanted activities. This includes the recording of all executed commands in log files (Transcription) as well as the newer Deep Scriptblock Logging.

The latter records all PowerShell actions in the event log. These entries can be encrypted using Protected Event Logging and thus be protected from prying eyes. Overall, PowerShell has a number of mechanisms that make malicious use much more difficult.



The event viewer presents only the encrypted entries, it cannot decode them.

Lee Holmes has compiled a table on Microsofts PowerShell-Teamblog that compares the security features of different programming languages and shells.

It shows that PowerShell offers more options than the others to prevent unwanted use. Of course, this does not provide an ultimate security, because resourceful minds always find ways to bypass the defense.

	Event Logging	Trans-cription	Dynamic Evalu-ation Logging	Encrypted Logging	App Whitelist-ing
Bash	No**	No*	No	No	Yes
CMD / BAT	No	No	No	No	Yes
JScript	No	No	No	No	Yes
LUA	No	No	No	No	No
Perl	No	No	No	No	No
PHP	No	No	No	No	No
PowerShell	Yes	Yes	Yes	Yes	Yes
Python	No	No	No	No	No
Ruby	No	No	No	No	No
sh	No	No	No	No	No
T-SQL	Yes	Yes	Yes	No	No
VBScript	No	No	No	No	Yes
zsh	No	No	No	No	No

	Antimalware Integration	Local Sand-boxing	Remote Sandboxing	Untrusted Input Tracking
Bash	No	No	Yes	No
CMD / BAT	No	No	No	No
JScript	Yes	No	No	No
LUA	No	No	Yes	Yes
Perl	No	No	Yes	Yes
PHP	No	No	Yes	Yes
PowerShell	Yes	Yes	Yes	No
Python	No	No	No	No
Ruby	No	No	No	Yes
sh	No	No	Yes	No
T-SQL	No	No	No	No
VBScript	Yes	No	No	No
zsh	No	No	Yes	No

** Feature exists, but cannot be enforced via policies*

***experimental*

However, to benefit from these protections, admins must invest more effort than just simply blocking powershell.exe. As a benefit they can keep PowerShell as a fully available system management tool which can even be fine-tuned to delegate tasks to standard users.

2 Restrict execution of scripts

2.1 Setting an execution policy

The execution of PowerShell scripts can be restricted by policies, by default it is blocked. While the execution policy set interactively by the admin can be overridden by any user, configuration via GPO is more sustainable. However, it still does not provide security against malicious users.

The main purpose of the *execution policy* is to protect users from accidentally running untrusted scripts. The default setting on a freshly installed Windows is *Restricted*, so that no user can start PowerShell scripts, not even an administrator.

2.1.1 Settings for the execution policy

Other possible values are:

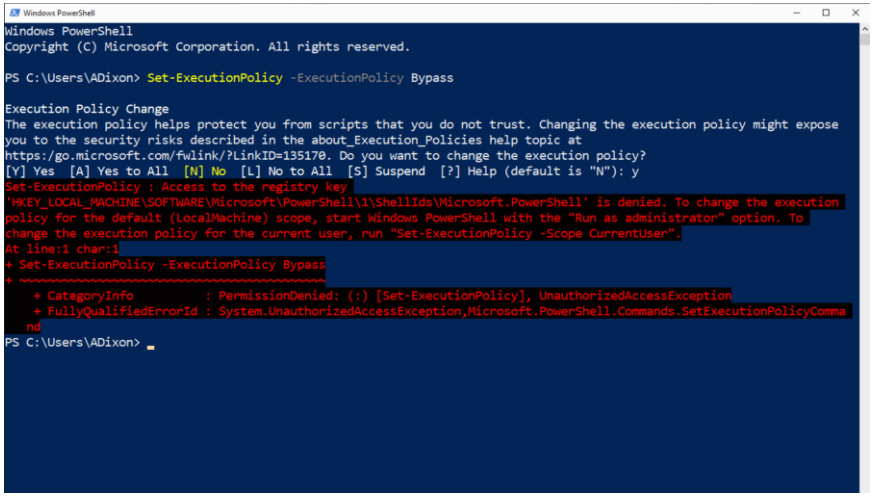
- AllSigned: Only signed scripts from a trusted publisher are executed, this also applies to locally created scripts.
- RemoteSigned: Scripts downloaded from the Internet must be signed by a trusted publisher.
- Unrestricted: All scripts are executed. For unsigned scripts from the Internet, you have to confirm each execution at the prompt.
- Bypass: No restrictions, warnings or prompts
- Undefined: Removes an assigned policy

2.1.2 Scope implicitly on LocalMachine

For example, if you want to change the default *Restricted* to *RemoteSigned* and enter the command

`Set-ExecutionPolicy RemoteSigned`

then it will fail if you have not opened the PowerShell session with administrative privileges.



Users without administrative rights cannot change the execution policy for the scope LocalMachine.

The reason for this lies in the validity area for the execution policy. If the scope is not explicitly specified, *Set-ExecutionPolicy* assumes *LocalMachine*. This would change the setting for all users on this machine, hence you need admin rights for this.

2.1.3 Overwrite PC-wide setting for a user

As is known from programming, a specific scope overrides a more general one. If you define the execution policy for the current user, it overwrites the one for the local machine. Therefore, any user can override a restrictive, system-wide setting as follows:

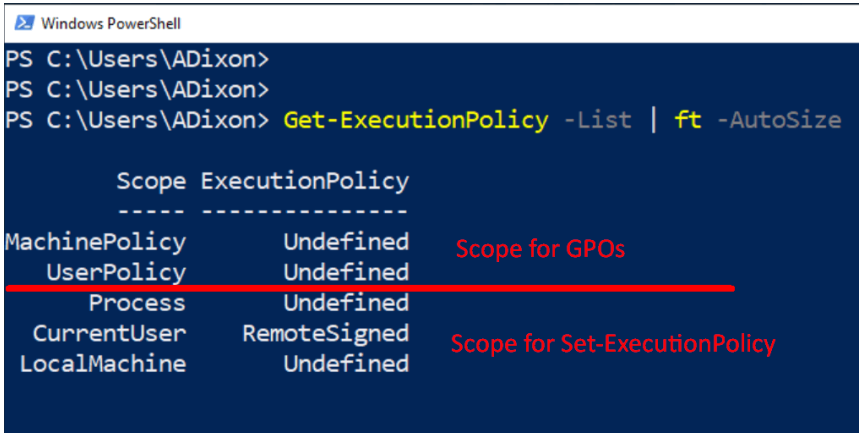
```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```

The scope *Process*, which affects the current session, is even more specific. The setting for this is not stored in the registry as usual, but in the environment variable $env:PSExecutionPolicyPreference. It is discarded at the end of the session.

2.1.4 Displaying policies for all scopes

The configuration of the execution policy for each scope can be displayed with:

```
Get-ExecutionPolicy -List | ft -AutoSize
```

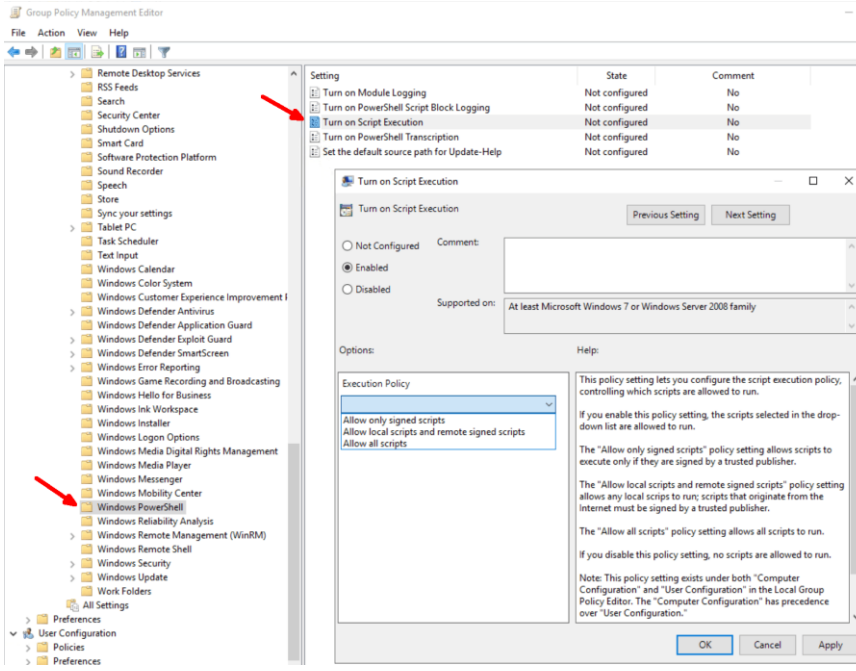


Scope of the PowerShell ExecutionPolicy

In addition to the *LocalMachine, CurrentUser*, and *Process* scopes described above, two others appear in the output of the cmdlet, namely *MachinePolicy* and *UserPolicy*. The values for these can only be set by using group policy.

2.1.5 Defining execution policy via GPO

The setting responsible for configuring the execution policy can be found for the computer and user configuration under *Policies => Administrative Templates => Windows Components => Windows PowerShell* and is called *Turn on Script Execution*.



GPO setting to configure the PowerShell execution policy

The execution policy configured in this way overrides the interactively defined values and also prevents an administrator from changing them on the command line. A bypass by invoking a new shell with

```
powershell.exe -ExecutionPolicy "Unrestricted"
```

does not work either, whereas this technique can be used to override a policy for *LocalMachine*. Furthermore, resetting to the *Undefined* value is only possible by deactivating the GPO.

A group policy can thus be used to specify which criteria scripts must meet in order to be allowed to run (this policy does not affect logon scripts, by the way). This prevents untrustworthy scripts from accidentally causing damage due to settings that are too lax.

2.1.6 No protection against malicious users

If a user decides to circumvent this policy, he simply copies the contents of a script to the ISE and runs it there. *RemoteSigned* allows unsigned scripts downloaded from the Internet to be started if you unblock the file using *Unblock-File*.

Another bypass consists of encoding the script in Base64 and transferring it to PowerShell.exe via the *EncodedCommand* parameter. To limit possible damage caused by such activities, it is recommended to use the Constrained Language Mode.

2.2 Signing PowerShell scripts

To ensure the authenticity of scripts, PowerShell is able to stamp them with a signature. You need a signature if you want to set policies that allow only trusted scripts to run. The required certificate can be issued by an AD-based CA for internally developed scripts.

By signing a script, its developer confirms that it originates from him and thus ensures that it has not been subsequently modified. Users who do not want to execute PowerShell code from an unknown source for security reasons can thus restrict the execution of scripts to certain manufacturers.

2.2.1 Restriction via execution policy, CLM, AppLocker

One mechanism for rejecting unsigned scripts is the execution policy. When set to *AllSigned*, both local scripts and scripts downloaded from the Internet must be signed. But this measure is not robust, because users can copy the content of the script to the prompt or to the ISE and start it there.

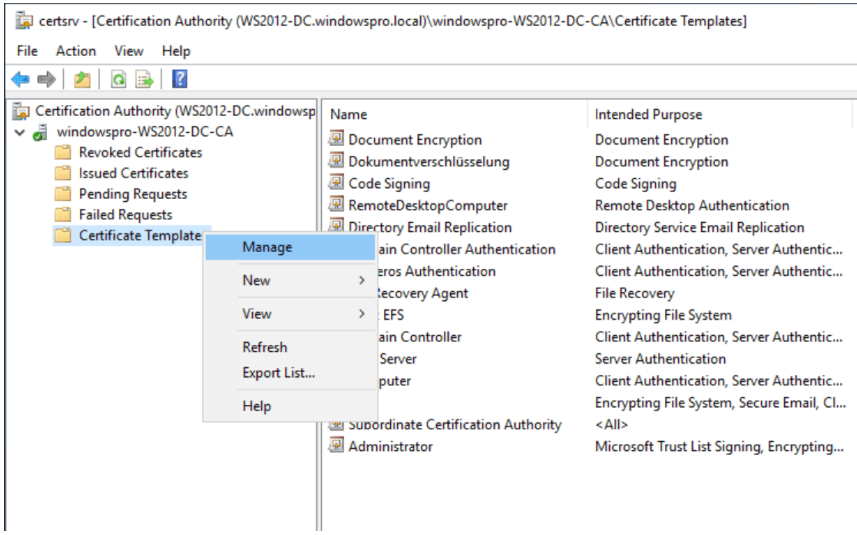
The Constrained Language Mode (CLM) offers more protection, because it only allows signed scripts to use the full functionality of PowerShell. Unsigned scripts, on the other hand, are denied access to features that have highly destructive potential.

Finally, solutions for whitelisting applications have the strongest effect in blocking untrustworthy scripts. For example, AppLocker can be used to restrict the execution to scripts from certain vendors.

2.2.2 Assign permissions to certificate template

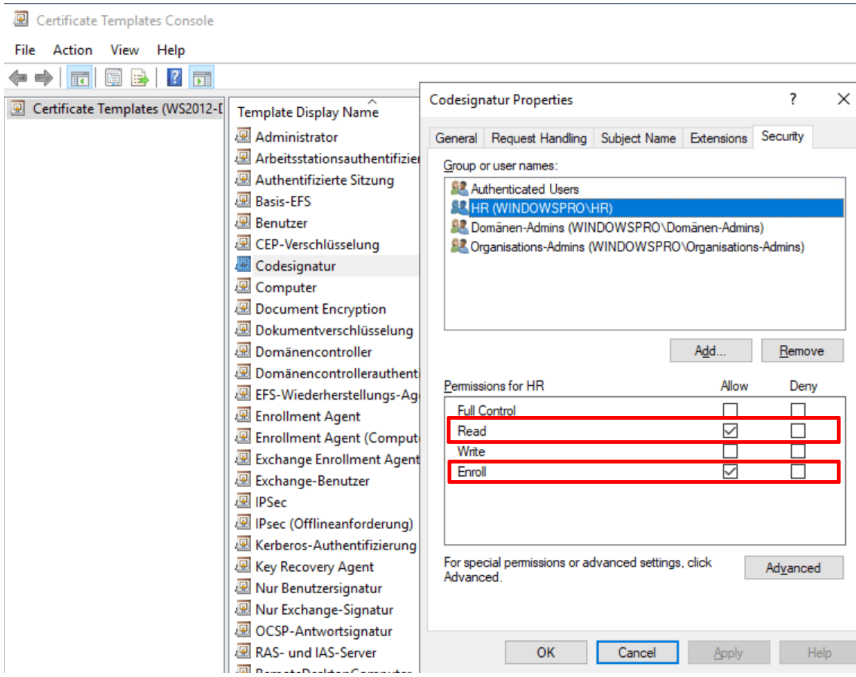
The first step is to make sure that the certificate template for code signing is accessible to users who want to request a certificate for their scripts. To

do this, open the MMC-based tool *Certification Authority* (certsrv.msc) and connect to the internal CA.



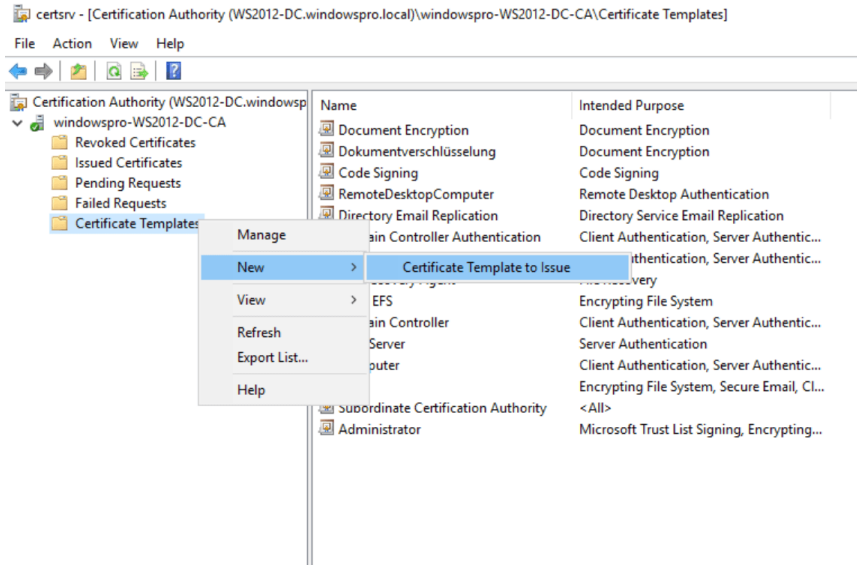
Open certificate templates from the MMC tool Certification Authority (certsrv.msc)

From the context menu of *certificate templates*, execute the *Manage* command. This opens the snap-in for certificate templates.



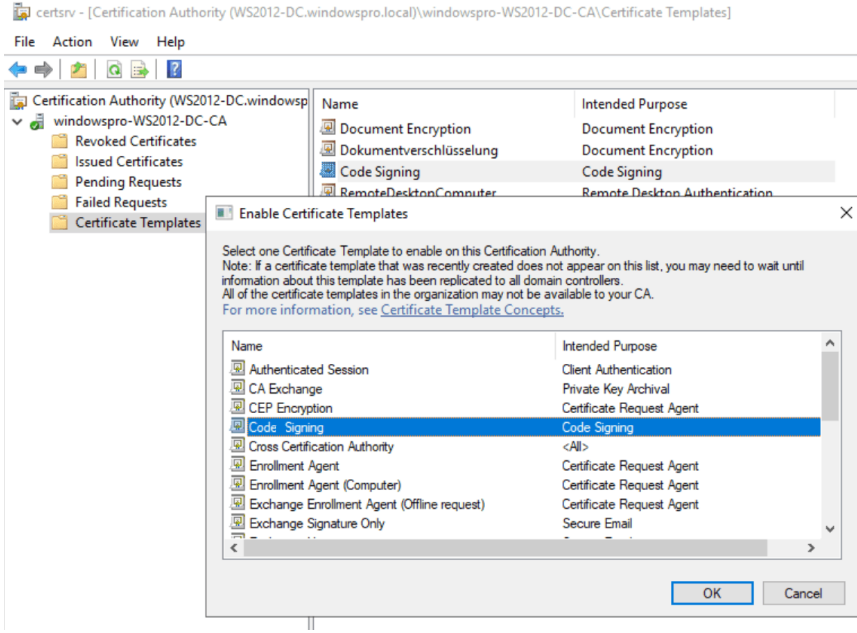
Assigning rights to the template for code signing

There you select *Properties* from the context menu of *Code signing* and switch to the *Security* tab. Next you add the group that should request certificates based on this template and grant it the *Read* and *Enroll* permissions.



Open the dialog for activating certificate templates

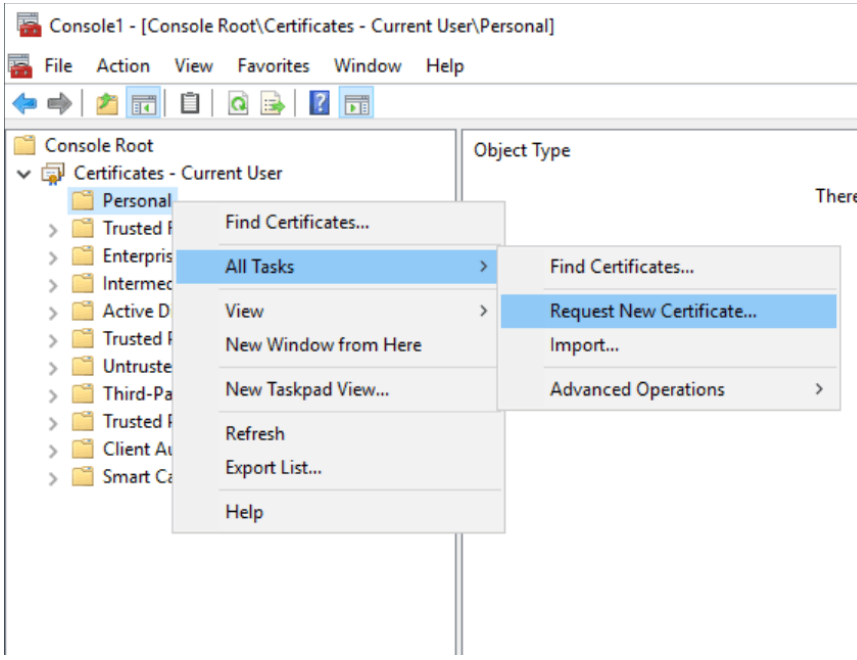
After confirming this dialog, return to certsrv.msc. From the context menu of certificate templates execute the command *New => Certificate Template to Issue.* In the following dialog you select *code signing* and close it with Ok.



Enabling the certificate template for code signing

2.2.3 Requesting a certificate for code signing

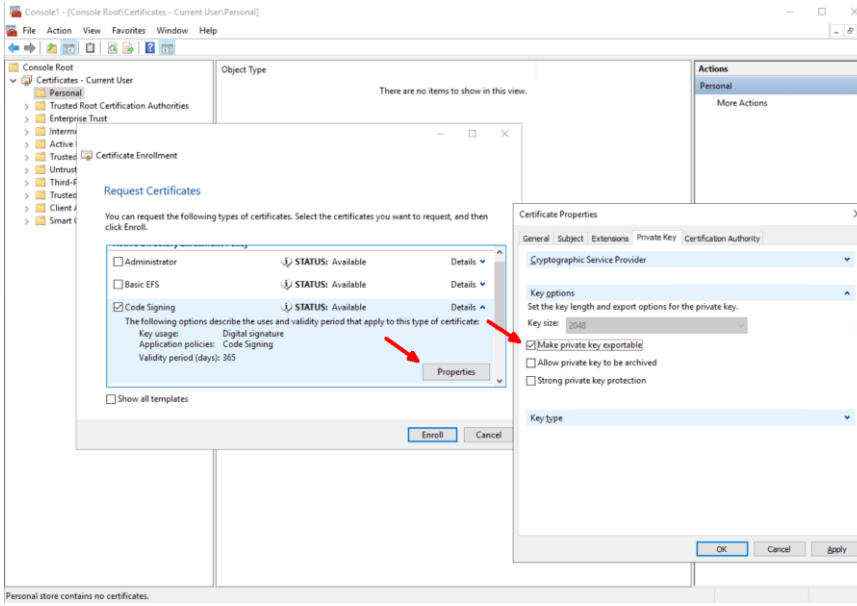
Now the developer of scripts can go ahead and request a certificate based on this template. To do this, he starts *mmc.exe* and adds the snap-in *certificates* from the *File* menu. For users who do not have elevated privileges, the tool automatically opens in the context of *Current User*.



Request a new code signing certificate

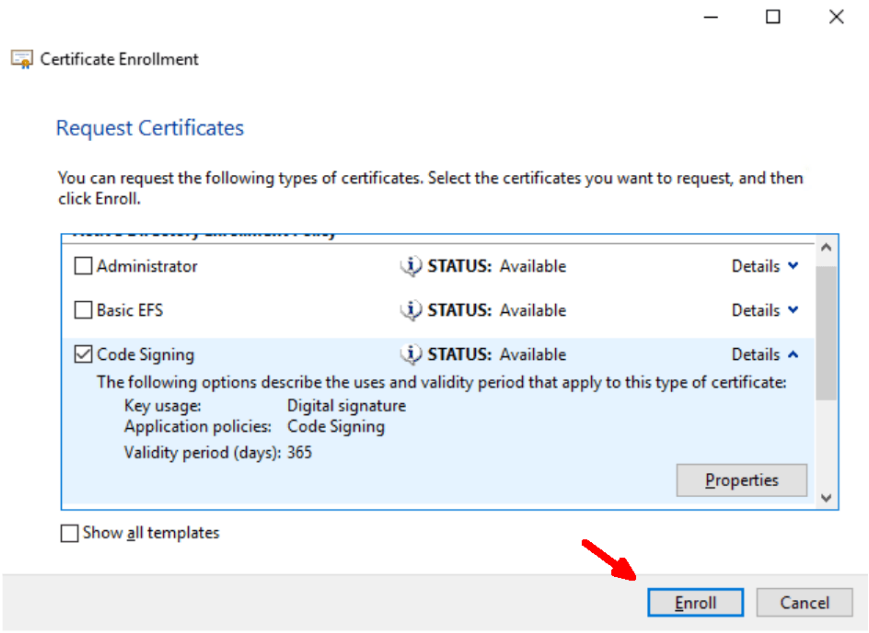
Here you right-click on *Personal* and then select *All Tasks => Request New Certificate*. This starts a wizard where you select the certificate enrollment policy in the first dialog (usually the default one for AD).

Then you select the template *Code Signing*, open its details and click on *Properties*. In the dialog that appears, enter the necessary data under *Subject* and switch to the *Private Key* tab to check the option *Make private key exportable*.



Select the code signing template and make the private key exportable

After confirming this dialog, back in the main window click on *Register*. Now the result of the operation is displayed and you can complete the process with *Enroll*.



Successful completion of the certificate request

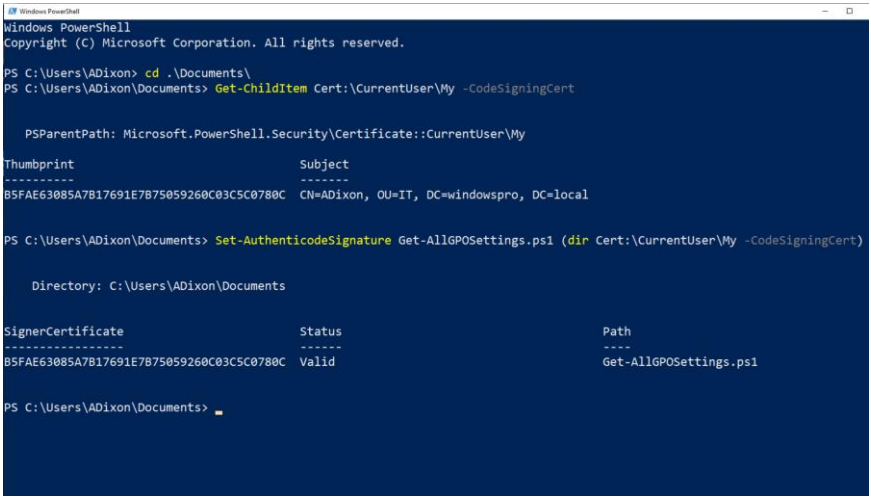
2.2.4 Signing a script

The certificate can now be found in the user's local store under *Personal => Certificates*. This can be displayed in PowerShell using the corresponding provider:

```
Get-ChildItem Cert:\CurrentUser\My -CodeSigningCert
```

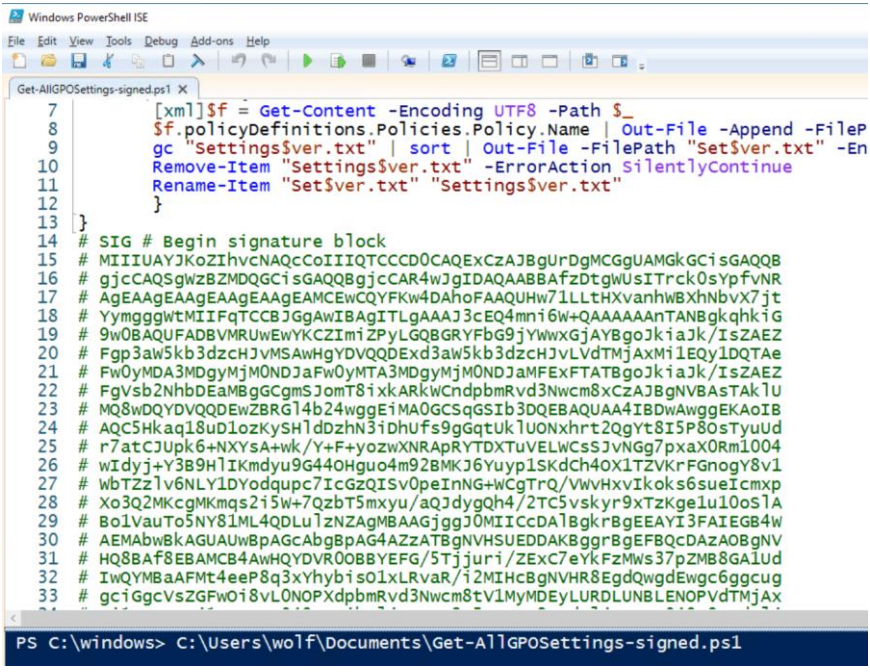
You can take advantage of this command used to specify the certificate when signing the script with *Set-AuthenticodeSignature*:

```
Set-AuthenticodeSignature myScript.ps1 `
(dir Cert:\CurrentUser\My -CodeSigningCert)
```



Signing a script by using the Set-AuthenticodeSignature cmdlet

PowerShell will insert the signature in Base64 format as a separate block at the end of the script.



PowerShell script after signing with a certificate

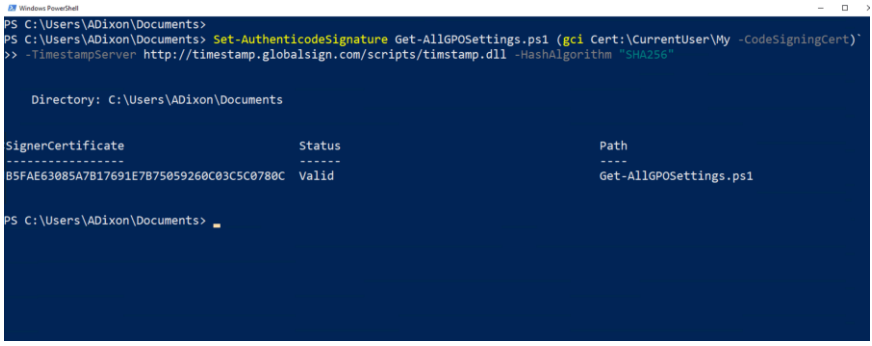
When the script is started for the first time on a computer after signing, the user must confirm the execution if the publisher is not considered to be trustworthy.

If you select the option *Always run*, this prompt will not appear in the future because the certificate is saved in the store. In this respect, PowerShell behaves just like a web browser or RDP client.

2.2.5 Marking the signature with a time stamp

After signing a script, PowerShell will refuse to execute it if you make even the slightest change to it. The only remedy is to re-sign the script.

The same applies when the certificate expires. In this case the script can also no longer be used. But you can prevent this by using a timestamp server when signing.



Signature with a time stamp

This example uses the free service of Globalsign:

```
Set-AuthenticodeSignature myScript.ps1 `
(gci Cert:\CurrentUser\My -CodeSigningCert)`
-TimestampServer http://timestamp.glob-
alsign.com/scripts/timstamp.dll `
-HashAlgorithm "SHA256"
```

This proves that the certificate was valid at the time of signing.

2.3 Reduce PowerShell risks with Constrained Language Mode

PowerShell is a powerful tool that can control almost all components of Windows and applications such like Exchange. It can therefore cause great damage in the hands of attackers. The constrained language mode blocks dangerous features and thus prevents their misuse.

By default, PowerShell operates in *Full Language Mode*, where all functions are available. This includes access to all language elements, cmdlets and modules, but also to the file system and the network.

2.3.1 Blocked Functions

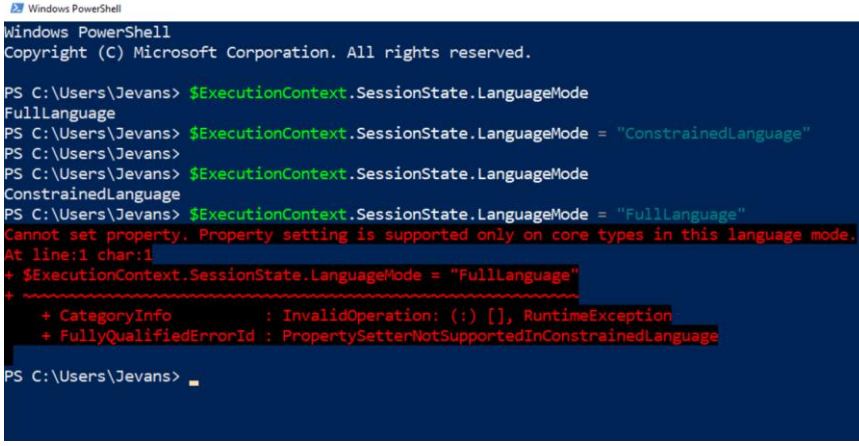
The ability to instantiate COM and .NET objects or to generate new data types (with add-type) that have been defined in other languages is particularly dangerous capability of PowerShell.

The constrained language mode blocks these features (except access to permitted .NET classes). It also prevents the declaration of classes, usage of configuration management with DSC, and XAML-based workflows (see Microsoft Docs for a complete list).

2.3.2 Enabling constrained language mode

A simple way to switch to Constrained Language Mode is to set the responsible variable to the desired value:

```
$ExecutionContext.SessionState.LanguageMode = `
"ConstrainedLanguage"
```

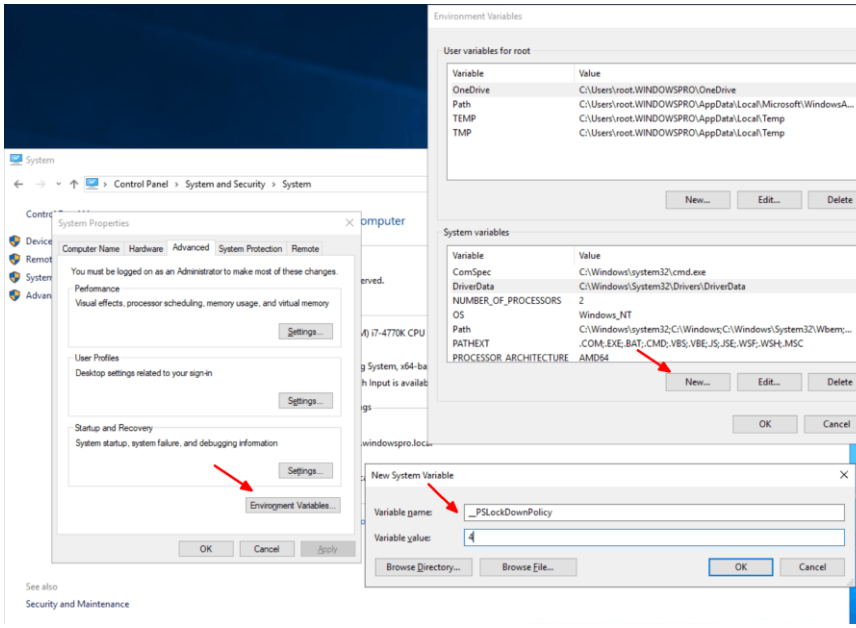


Displaying and changing the Language Mode via the variable $ExecutionContext.SessionState.LanguageMode

It is obvious that setting this variable does not provide any real protection. You may not be able to change it back to *FullLanguage* in the same session, but a new PowerShell session will again offer the full range of languages features.

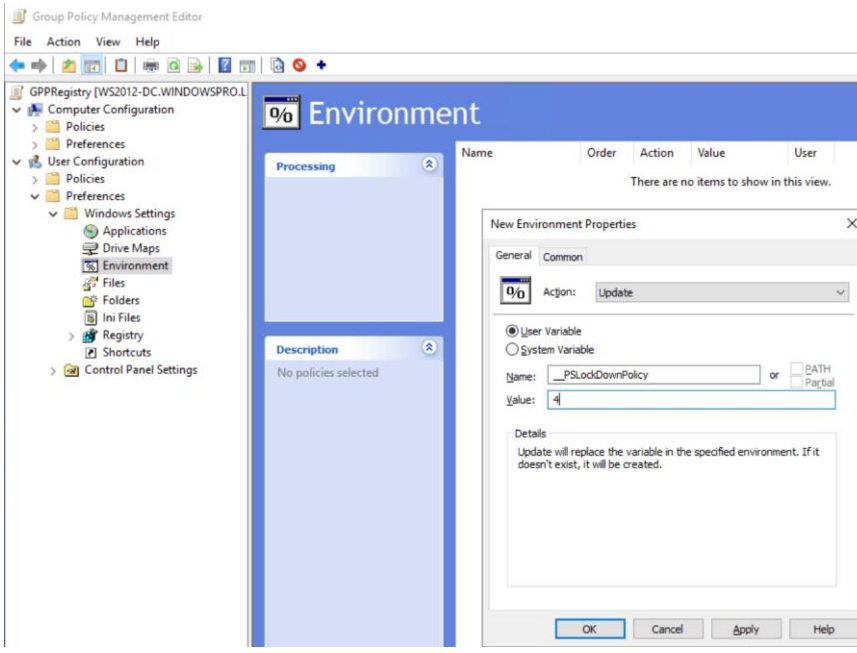
2.3.3 Switching to restricted mode with environment variable

Less easy to overcome is the (undocumented) system environment variable *__PSLockDownPolicy*, if you set it to the value 4. As a result, PowerShell, regardless of whether it's just a command line or the ISE, will start in restricted mode.



Setting environment variable __PSLockDownPolicy interactively

In centrally managed environments you will probably set the system variable using group policies preferences.



Setting environment variable __PSLockDownPolicy via GPO

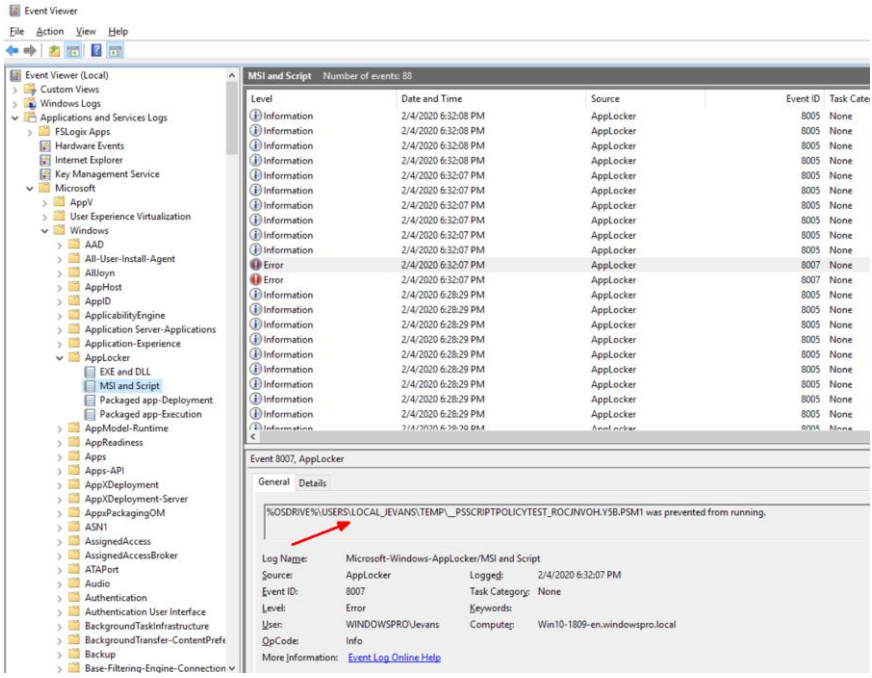
A disadvantage of this procedure is that it always affects all users of a computer, including administrators. However, administrators may temporarily remove the environment variable until the GPO becomes effective again. But this is quite cumbersome and definitely not a good solution.

Furthermore, when used this way, it is not a security feature supported by Microsoft and it is relatively easy to circumvent, as shown by Matt Graeber in this Tweet. Nevertheless, it might thwart most opportunist attacks.

A strict enforcement of the constrained language mode on a local computer thus requires the use of a software execution restriction such as AppLocker or Windows Defender Application Control. In a remote session, however, it can be enforced via a Session Configuration.

2.3.4 Automatic detection of an execution constraint

Since version 5, based on script rules PowerShell recognizes automatically whether it should switch to constrained language mode. To do so, it creates a module and a script (with a name following the pattern __PSSCRIPTPOLICYTEST_LQU1DAME.3DD.PS1) under $env:temp and tries to execute them. If AppLocker or another tool blocks this attempt, PowerShell will start in restricted language mode.



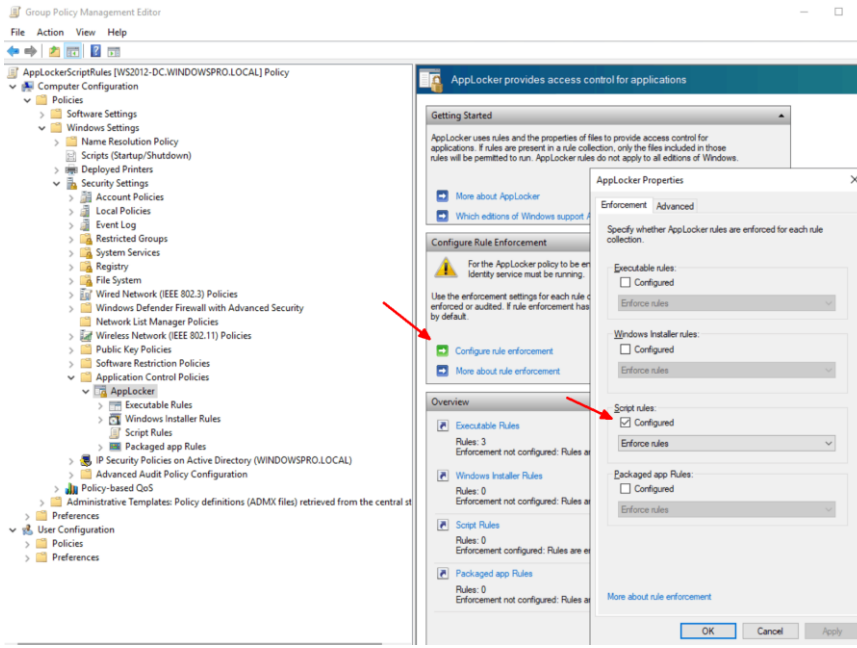
The event log shows whether the execution of the test scripts was successful or if it has failed.

The effect of this mechanism can easily be seen in AppLocker's event log. AppLocker logs the creation and execution of these test files with the ID

8005 (success) or 8007 (execution blocked) under *Applications and Services Log => Microsoft => Windows => AppLocker => MSI und Script*.

2.3.5 Configuring AppLocker

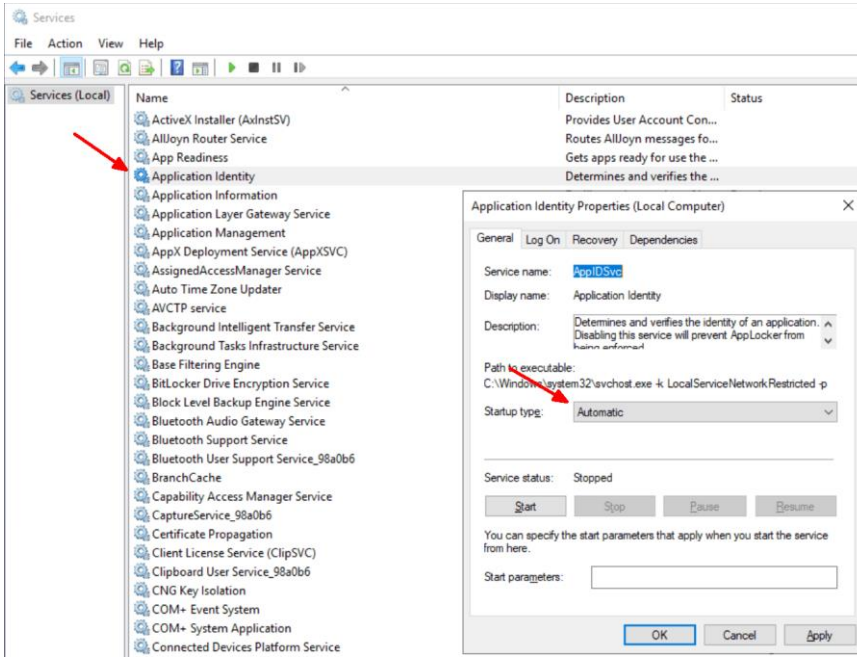
If you use AppLocker for this task, you have to create a new GPO and then edit it in the GPO editor. There you navigate to *Computer Configuration => Policies => Windows Settings => Security Settings => Application Control Policies => AppLocker* and follow the link *Configure rule enforcement*. In the dialog that appears, you then activate the option *Script rules*.



Enabling rule enforcement for scripts in AppLocker

In order for AppLocker to block applications on the target systems, the service named *Application Identity* must be running. It is not active by default and does not start up when the system is booting. You can change it to start type *Automatic* either interactively using the MMC snapin *services* or from the command line:

```
sc config AppIDSvc start=auto
```

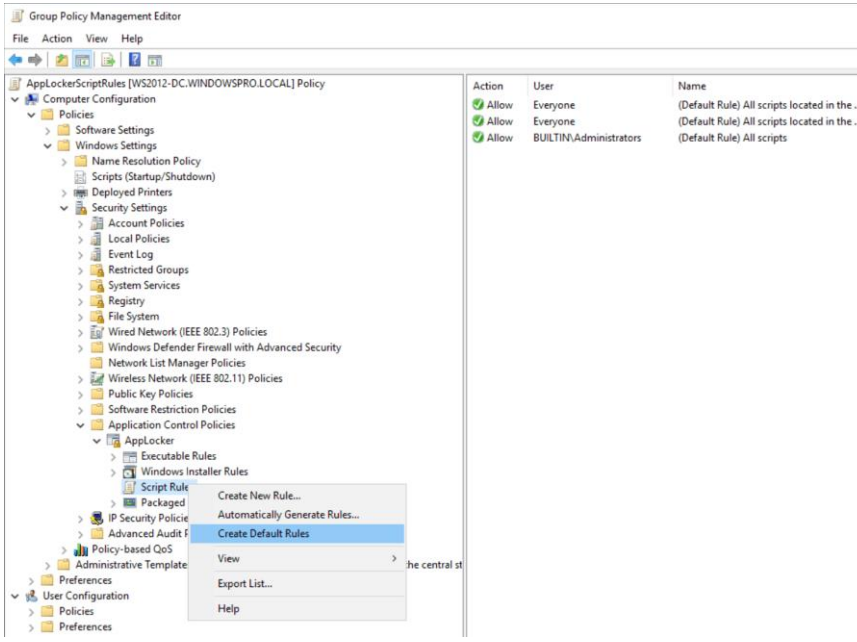


Setting the start type for the Application Identity service to automatic

For a central management of this Windows service, the use of Group Policy is recommended.

2.3.6 Defining rules

Finally it is necessary to define rules that block the start of scripts in the Temp directory. To do this, simply switch to *Script Rules* below AppLocker and select *Create Default Rules* from the context menu.



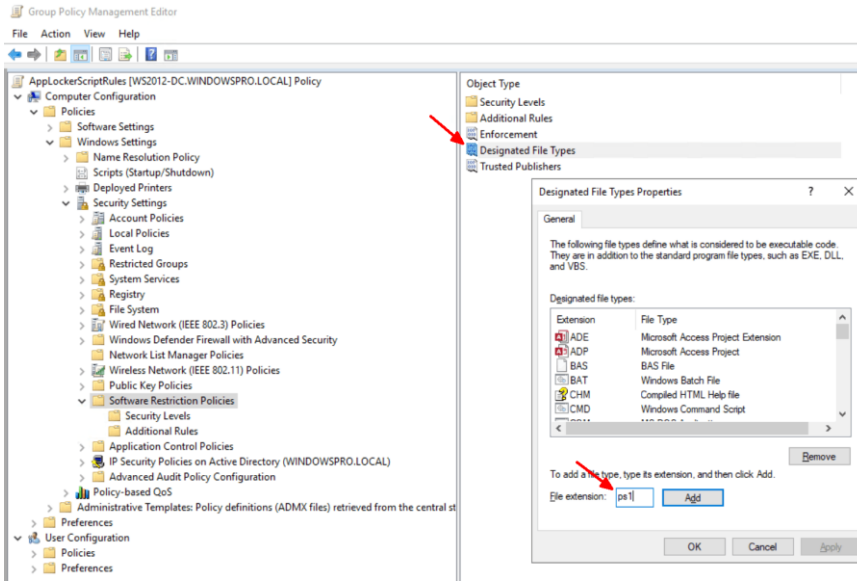
Creating default rules for scripts in AppLocker

They allow standard users to execute scripts only from the *Windows or Program Files* directories, i.e. in locations where users cannot store any files themselves. Administrators are explicitly exempted from this restriction by a separate rule.

2.3.7 Activating Constrained language mode via SRP

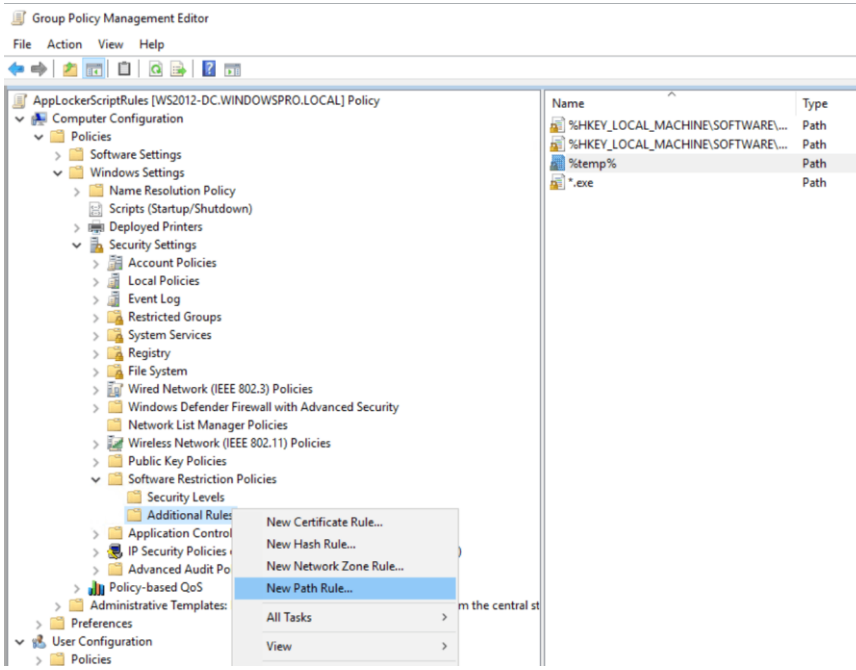
AppLocker is an exclusive feature of the Enterprise and Education editions. Therefore, the Pro edition can use the Software Restriction Policies (SRP) instead.

Again, you just have to ensure that the two test scripts cannot be executed in the %temp% directory. To do this, create a GPO and open it in the editor and navigate to *Computer Configuration* => *Policies* => *Windows Settings* => *Security Settings* => *Software Restriction Policies.*



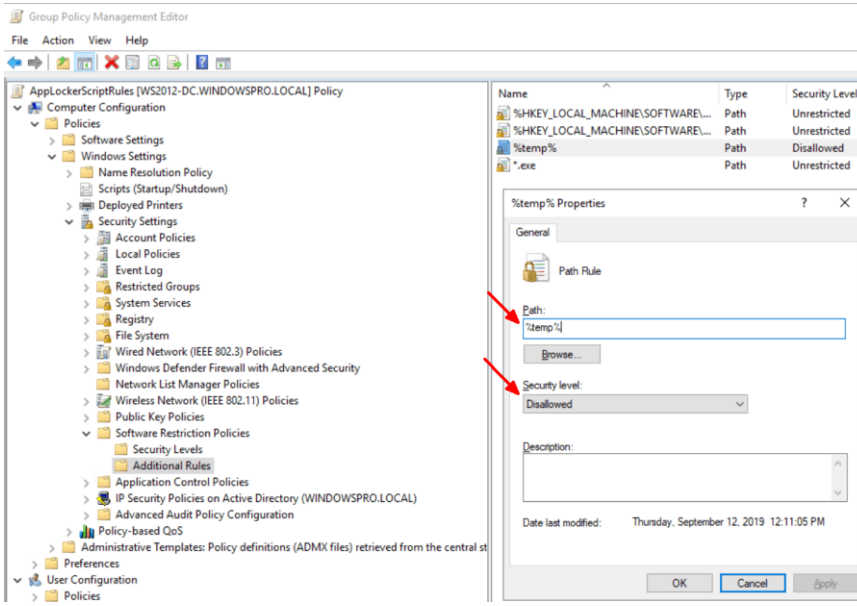
Enter file extensions for PowerShell in the Software Restriction Policies.

Here you create a new policy and in the first step you add the extensions *ps1* and *psm1* to the list of the designated file types.



Creating a New Path Rule for the software restriction

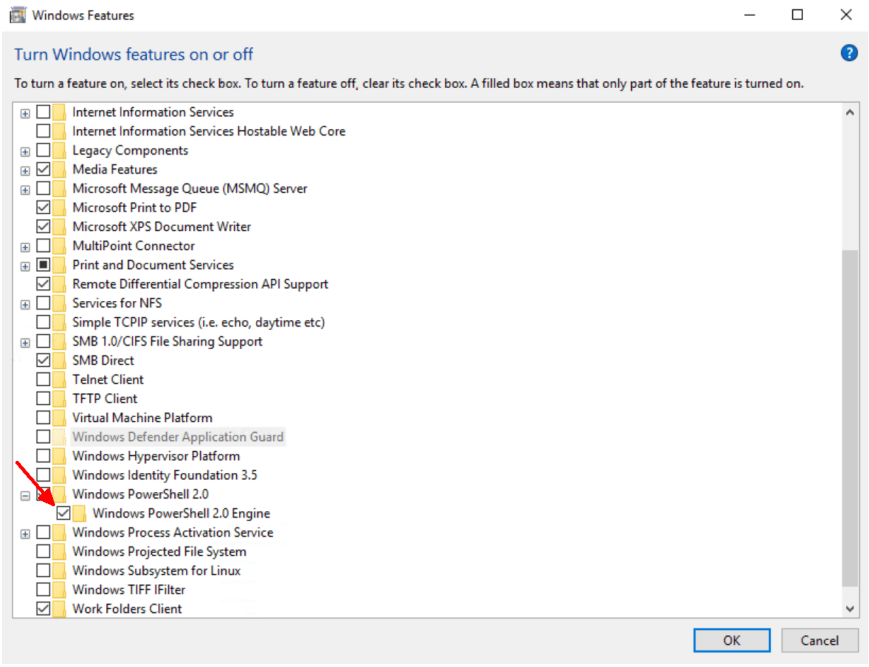
Then you create a *New Path Rule* under *Additional Rules*. Here you enter %temp% as the *Path* and leave the setting for *Security level* set to *Disallowed*.



Defining the path rule for the Temp directory

2.3.8 Prevent PowerShell 2.0 circumvention

Regardless of whether you choose the environment variable, AppLocker, or Software Restriction Policies, you will need to remove PowerShell 2.0 from the machines where you want to enforce the constrained language mode.



PowerShell 2.0 is an optional feature starting with Windows 8 and Server 2012 and is enabled by default.

It has only been introduced with PowerShell 3.0 and can easily be bypassed by a hacker switching to an older version. All he needs to do is to enter the command:

```
powershell.exe -version 2.0
```

You can check whether this old version is still activated on a PC by entering:

```
Get-WindowsOptionalFeature -Online `
-FeatureName MicrosoftWindowsPowerShellV2
```

However, you can only uninstall it on Windows 8 and Server 2012 or later, where PowerShell 2.0 is an optional feature.

3 Secure communication

3.1 Installing OpenSSH on Windows 10 and Server 2019

Windows Server 2019 includes OpenSSH as an optional feature for the first time, thus simplifying installation and configuration. However, errors in the earlier builds of the operating system prevent a successful activation of the SSH server. In WSUS environments OpenSSH has the same problems as RSAT.

The porting of OpenSSH to Windows makes it easier to manage heterogeneous environments. Linux computers can be remotely administered via SSH from Windows, and thanks to the new OpenSSH server, the reverse is now also possible. In addition, PowerShell Core supports remoting via SSH, even between different OSes.

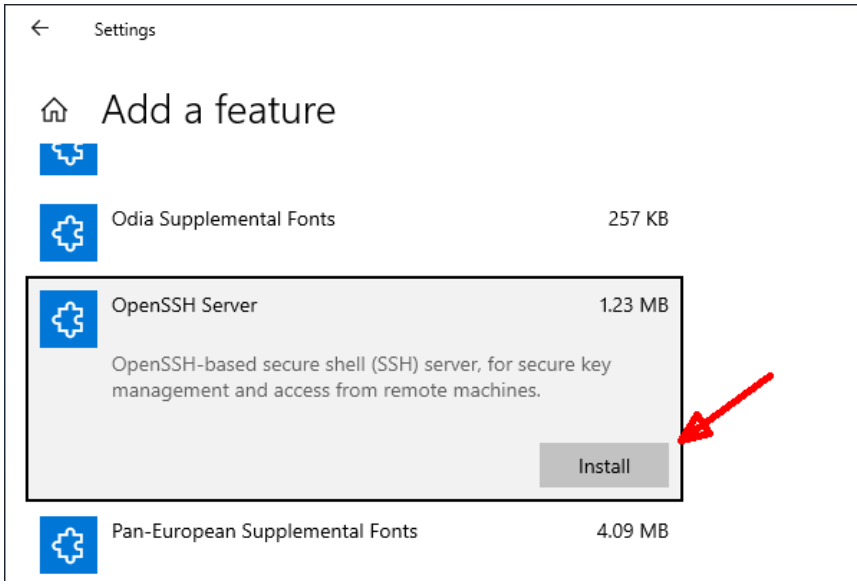
3.1.1 OpenSSH server not included in the operating system

One would expect that a system component with such strategic importance is delivered as part of the operating system and can be installed as a feature via the Server Manager or PowerShell.

However, Microsoft has decided to provide OpenSSH as an optional feature (also called "Feature on Demand"). This unifies the installation between client and server OS. The following description therefore also applies to Windows 10 from Release 1803 onwards.

3.1.2 Installation via GUI

To install OpenSSH server, start Settings, then go to *Apps => Apps and Features => Manage Optional Features*. As you can see from the list of installed components, the SSH client is already installed by default. The server, on the other hand, you need to add using the Add Features option.



Installing the OpenSSH Server via the Settings App

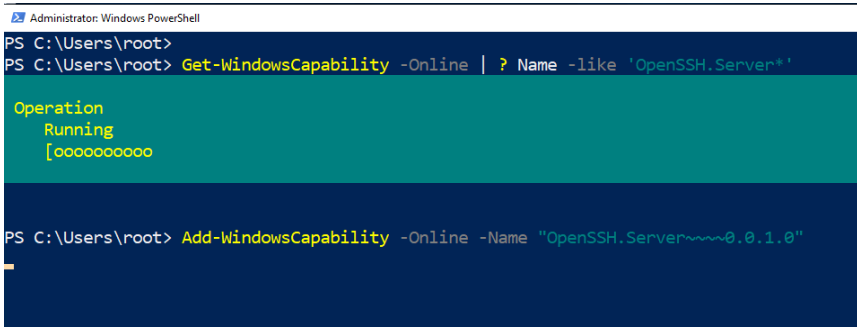
In the list above, select OpenSSH server and click on the Install button that appears. Windows will now download the required files over the Internet. If an error occurs, you will not receive a message from the Settings App, but it will simply jump back to the list of features.

3.1.3 Adding an OpenSSH-Server via PowerShell

In contrast, PowerShell provides more transparency. To find the exact name of the required package, you enter the following command:

Get-WindowsCapability -Online | ? name -like *OpenSSH.Server*

Finally you add the name shown to *Add-WindowsCapability*.



Adding an OpenSSH Server via PowerShell

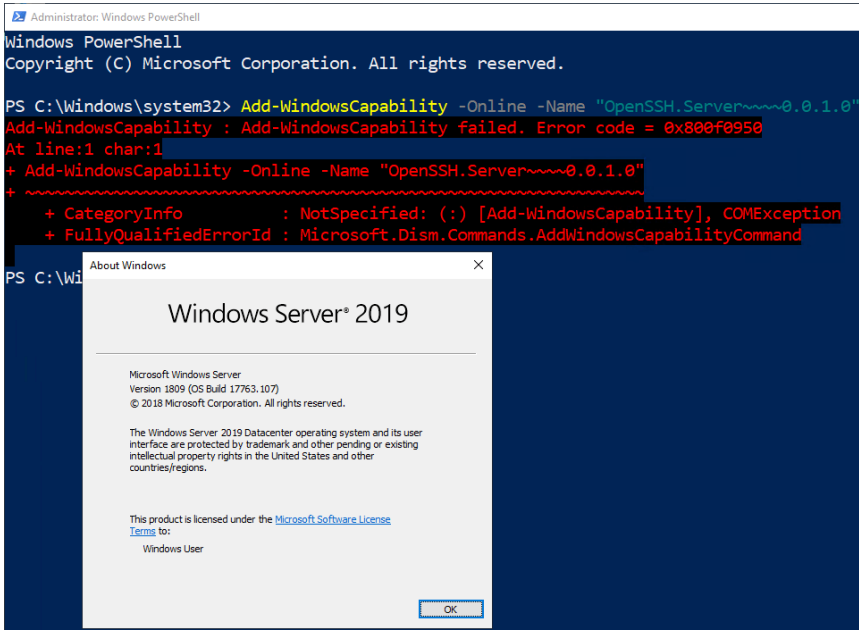
Alternatively, you can pass on the output via a pipe:

```
Get-WindowsCapability -Online |
where name -like *OpenSSH.Server* |
Add-WindowsCapability -Online
```

3.1.4 Faulty Builds

There are at least two reasons why you may encounter problems here. If the build of the system is older than 17763.194, then you will see the error

```
Add-WindowsCapability failed. Error code = 0x800f0950
```



The installation of OpenSSH Server fails on earlier builds of Windows Server 2019.

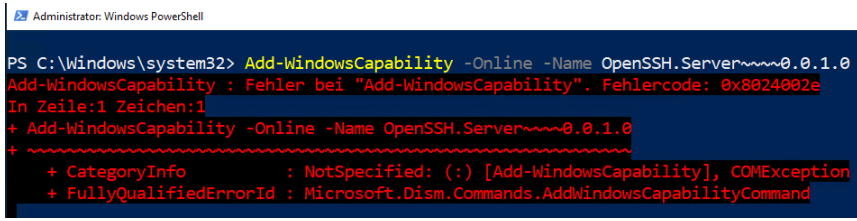
In this case you need a current cumulative update to fix the problem (it is documented here: bit.ly/3kCi0Pv).

3.1.5 Problems with WSUS

A further hurdle arises if the server, which is usually the case, is updated via WSUS. Microsoft delivers features on demand bypassing WSUS, so you don't get them via the internal update server.

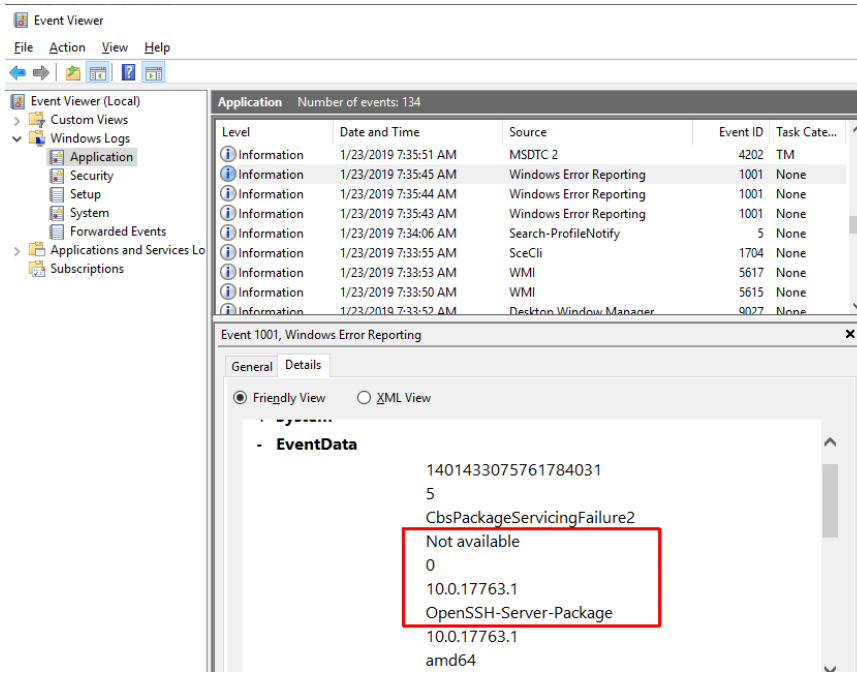
Therefore, it is not unlikely that PowerShell will present the following error here:

```
Error with "Add-WindowsCapability". Error code: 0x8024002e
```



Error while installing OpenSSH as an optional feature in WSUS environments

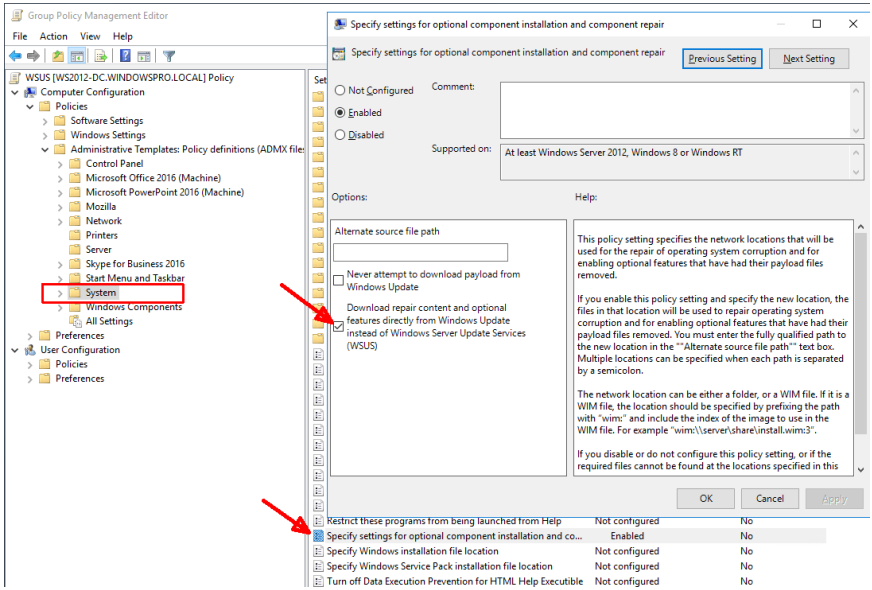
In the eventlog you will then find an entry with ID 1001 stating that the OpenSSH-Server-Package is not available.



Eventlog entry when adding OpenSSH server as optional component in a WSUS environment

As with the RSAT, a remedy is to allow Windows to load optional features directly from Microsoft Update via group policy. The Setting is called *Specify settings for optional component installation and component repair* and

can be found under *Computer Configuration => Policies => Administrative Templates => System*.



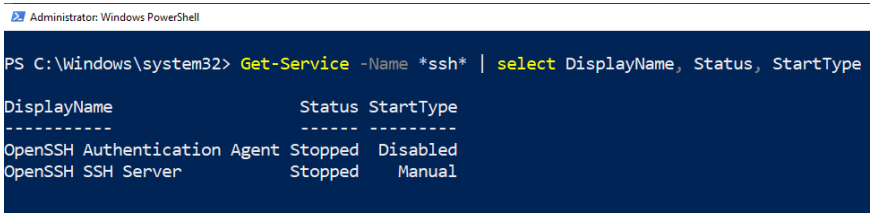
Allowing WSUS clients to access Windows Update using Group Policy.

At the same time, you must ensure that neither the setting *Do not connect to Windows Update Internet locations* nor *Remove access to use all Windows Update features* is in effect.

The latter may have been enabled to prevent users from manually downloading feature updates. This primarily affects Windows 10 rather than the server.

3.1.6 Activating SSH-Server

OpenSSH Server installs two services which are not yet running and whose startup type is manual and disabled. If you want to use SSH regularly, you will want to start the services automatically.



Displaying the Startup Type and Status of SSH Services with PowerShell

This can be configured via the GUI services, but the fastest way is using PowerShell:

```
Set-Service sshd -StartupType Automatic
```

```
Set-Service ssh-agent -StartupType Automatic
```

To put the SSH server into operation immediately, you must also start the two services manually:

```
Start-Service sshd
```

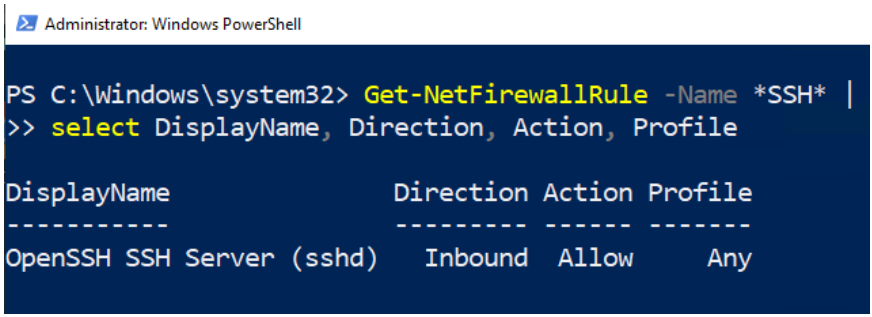
```
Start-Service ssh-agent
```

This command

```
Get-Service -Name *ssh* |
select DisplayName, Status, StartType
```

is used to check whether the settings for the two services match and whether they were started successfully. Now you can check if the firewall rule for incoming SSH connections has been properly activated:

```
Get-NetFirewallRule -Name *SSH*
```



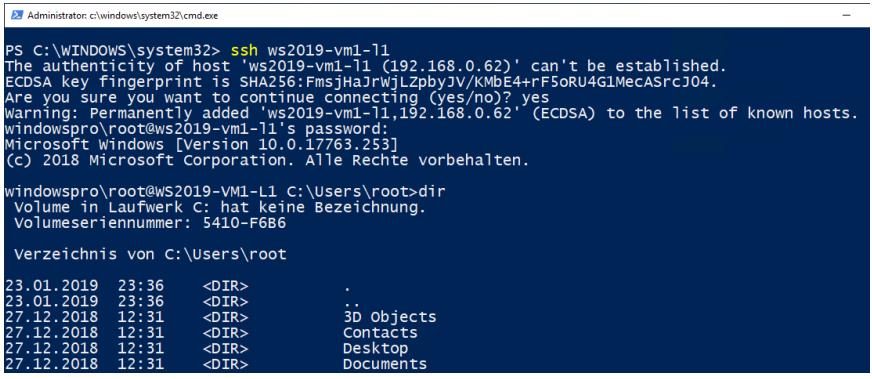
Checking Firewall-Rule for SSH

3.1.7 Testing the connection

If this condition is also fulfilled, then the connection test is good to go. From a Windows 10 PC or a Linux computer you can connect to the freshly configured server:

```
ssh <Name-of-Server>
```

This will direct you at the old command prompt, but you can also start PowerShell there.



Establish connection to freshly installed SSH server

Finally, you should consider whether you would like to use public key authentication for security reasons. This also increases user comfort because you no longer have to enter a password.

3.2 PowerShell remoting with SSH public key authentication

One of the advantages of PowerShell remoting via SSH over WinRM-based remoting is that you can work with public key authentication. This makes remote management of Windows machines that are not members of an Active Directory domain convenient and secure.

If you work with WinRM in an environment without Active Directory, things get quite messy and inconvenient if security matters to you. You have to switch from the default HTTP to the HTTPS protocol, deal with SSL/TLS certificates and with trusted hosts.

Remoting over SSH, which has been introduced with PowerShell 6, doesn't require public key authentication to work. Instead, username and password are also accepted.

The main downside is that you then have to enter your Windows password every time you connect to a remote machine. That might be okay for interactive sessions with *Enter-PSsession*, but if you want to run your scripts remotely via *Invoke-Command,* it could be a problem.

Moreover, public key authentication improves security because it works conveniently without using passwords. Thus, it makes sense to invest a little more time and configure PowerShell remoting for public key authentication.

3.2.1 Local configuration

The first thing you have to do is create the private and the public key, which you can do by simply running the *ssh-keygen* command. By default,

the command saves the key pair in the .ssh folder in your user profile. *id_rsa* is the private key, and *id_rsa.pub* is the public key.

If you want to work without a passphrase, you can just hit Enter twice. However, I recommend using a passphrase because if someone gets access to your private key, this will compromise all your remote machines.

Thanks to the ssh-agent, you don't have to enter the passphrase whenever you connect to a remote machine. The ssh-agent runs as a service and securely stores your private key. At a PowerShell console, you can start the ssh-agent this way:

```
Start-Service ssh-agent
```

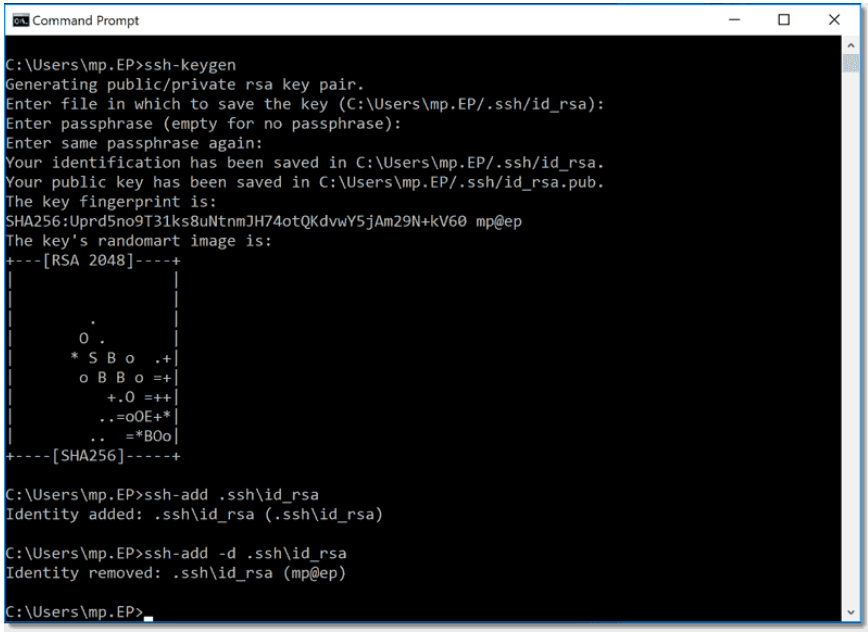
If you want the service to start automatically after a restart, you can use this command:

```
Set-Service ssh-agent -StartupType Automatic
```

To add your private key to the ssh-agent, you have to enter this command:

```
ssh-add <path to private key>
```

You will have to enter your passphrase here once. After that you can remove your private key from the .ssh folder and store it in a safer place.



Creating a key pair, adding the private key to the ssh agent and removing it again

If you later want to remove the private key from the ssh-agent, you can do it with this command:

```
ssh-add -d ida_rsa
```

Note that this requires that you provide the SSH key. In case you have lost your private key, you can remove all private keys from the ssh-agent:

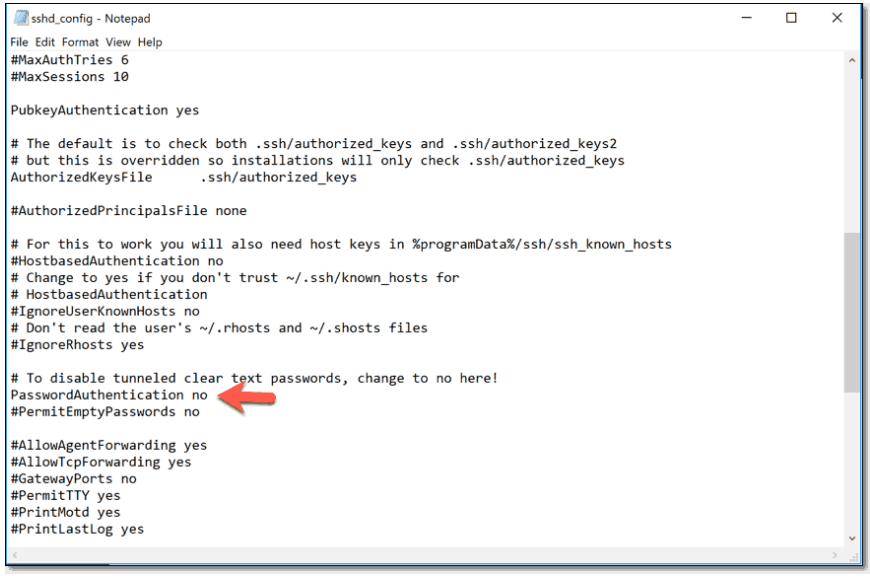
```
ssh-add -D
```

3.2.2 Remote configuration

Next, you have to copy the contents of the public key file id_rsa.pub to the remote host. Just paste it to the authorized_keys file in C:\Users\<your user name\.ssh\.

The public key for SSH (contents of id_rsa.pub)

By default, public key authentication is enabled in OpenSSH. However, I recommend disabling password authentication for security reasons. If an attacker compromises your Windows password, he can connect to the remote host even without your private key and passphrase.



Disabling password authentication for SSH

To disable password authentication, launch Notepad with admin rights and then open sshd_config in C:\ProgramData\ssh\. Add

```
"PasswordAuthentication no"
```

to the file and save it. You have to restart the ssh service to apply the changes. You can do this at a PowerShell console with admin rights:

```
Restart-Service sshd
```

3.2.3 Connecting with public key authentication

You are now back onto your local host and ready to test your connection.

At a PowerShell 6 or 7 console, simply enter this command:

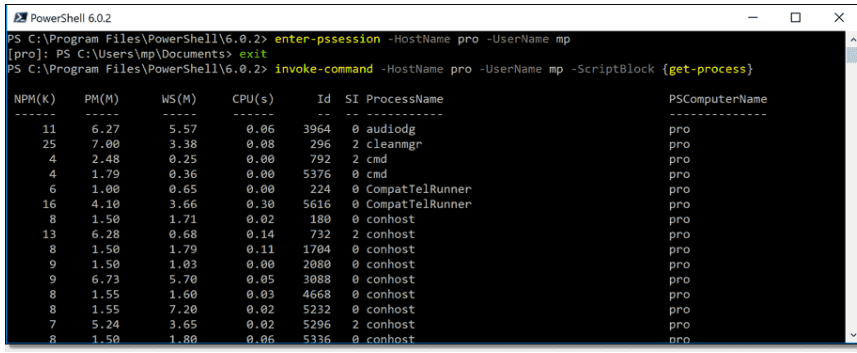
```
Enter-PSession -HostName <remote host> `
-UserName <user name on the remote computer>
```

The *HostName* parameter ensures PowerShell will connect via SSH instead of WinRM. Note that your user name on the remote computer doesn't have to be same if you use the *UserName* parameter. If you omit this parameter, PowerShell will take your current logon name on the local computer.

Notice you have to enter neither the Windows password nor the passphrase for the private key.

Invoke-Command works in just the same way:

```
Invoke-Command -HostName <remote hosts> `
-UserName <user name on the remote computer> `
-ScriptBlock {get-process}
```



PowerShell remoting via SSH transport and public key authentication

You can also connect with any SSH client. OpenSSH comes with a simple SSH client you can launch from the command prompt:

```
ssh <user name on the remote computer>@<remote host>
```

Just for the sake of completeness, if you didn't store your private key in the ssh-agent, you can still work with public key authentication. If the private key is located in the .ssh folder of your user profile, OpenSSH will automatically find the key. If you stored the key in another location, you have to pass the private key.

With the ssh client you can use the -i parameter:

```
ssh -i <path to private key>id_rsa <user name on the remote host>@<remote host>
```

Enter-PSsession and *Invoke-Command* have the *-IdentityFilePath* parameter for this purpose:

```
Enter-PSession -HostName <remote host> `
-UserName <user name on the remote host> `
-IdentityFilePath <path to private key>id_rsa
```

As mentioned above, I don't recommend working this way because it requires storing your private key in clear text on your local computer. Even

if you use a passphrase, it is more secure to work with the ssh-agent because you are safe from keyloggers and other password stealing methods.

3.3 Creating a self-signed certificate

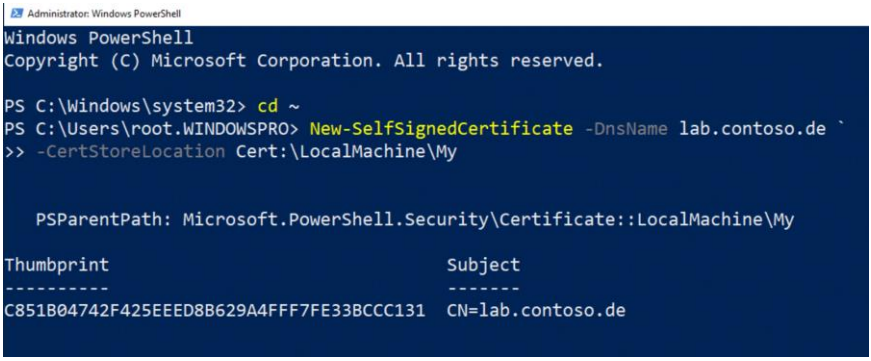
While back in Windows XP tools like makecert.exe were needed to issue self-signed certificates, since Windows 8 and Server 2012 PowerShell can take over this task with its cmdlet *New-SelfSignedCertificate*. The certificates can be used for client and server authentication or for code signing.

Self-signed certificates are typically used in lab or other small environments where you don't want to set up a Windows domain or an independent certificate authority. The issuer and user are then usually the same person or belong to a small group.

3.3.1 Creating a certificate with default values

To issue a SSL certificate, the cmdlet *New-SelfSignedCertificate* requires only very few parameters. A basic command in an administrative session might look like this:

```
New-SelfSignedCertificate -DnsName lab.contoso.de `
-CertStoreLocation Cert:\LocalMachine\My
```



Creating a self-signed SSL certificate with New-SelfSignedCertificate based on the default settings.

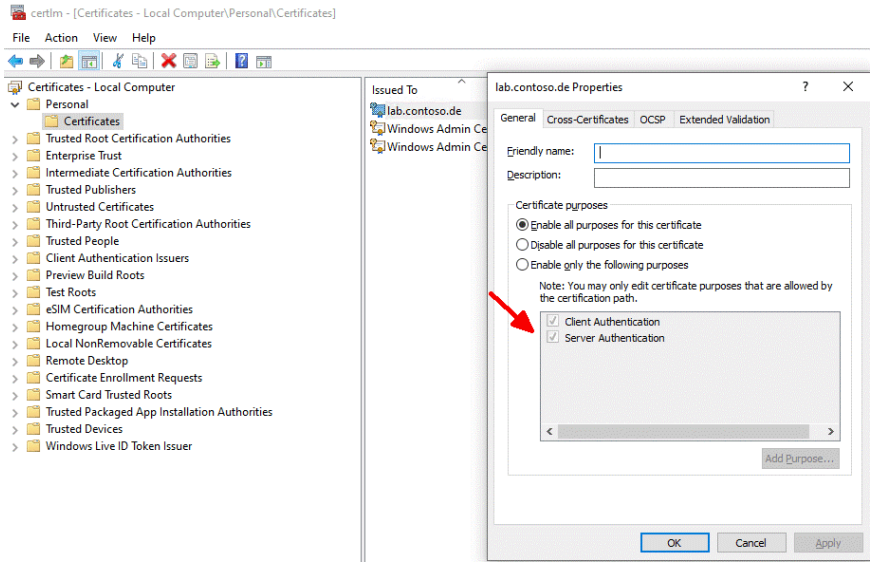
This command creates a new certificate under *My* in the store for the local machine, with the subject set to "lab.contoso.de".

Using the command

```
dir Cert:\LocalMachine\my\<thumbprint-of-certificate> |
fl -Property *
```

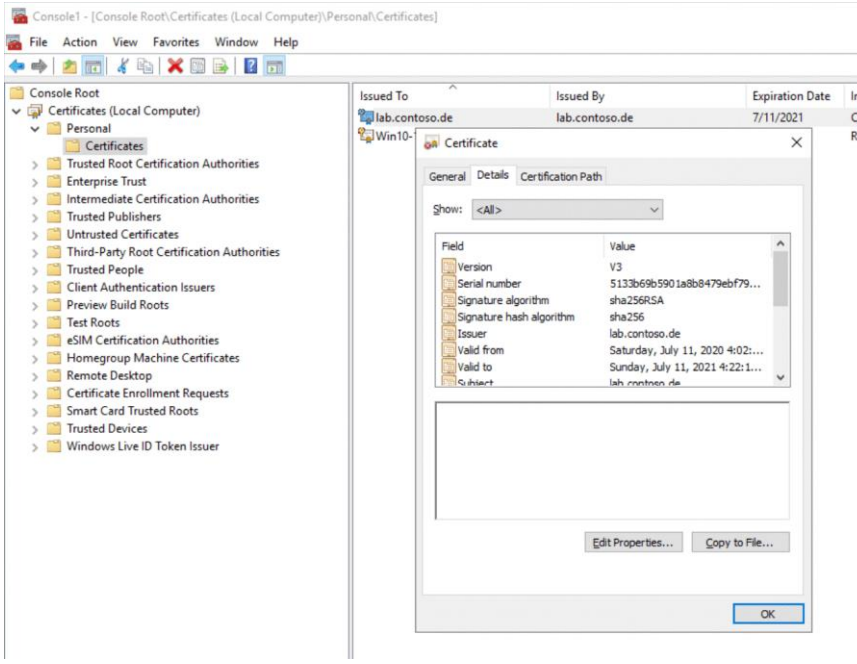
you can see that the new certificate has, among other things, the following default properties:

- EnhancedKeyUsageList: {client authentication(1.3.6.1.5.5.7.3.2), server authentication (1.3.6.1.5.5.7.3.1)}
- NotAfter: 22.03.2020 18:52:22
- HasPrivateKey: True
- Issuer: CN=lab.contoso.de
- Subject: CN=lab.contoso.de



If the certificate is generated with the default values, it will be suitable for client and server authentication.

Without specifying a type in the call to *New-SelfSignedCertificate*, the certificate is suitable for client and server authentication. Furthermore, it is valid for 1 year and has a private key which is also exportable as shown by *certutil*.



Displaying the properties of the new certificate in the MMC certificate snap-in.

The cmdlet issues a SAN certificate when you use the *DnsName* parameter. There you specify the subject alternative names as a comma-separated list. The first of them also serves as the subject as well as the issuer if you do not use a certificate to sign the new certificate by using the *signer* parameter.

You may also specify wildcards following the pattern

```
New-SelfSignedCertificate -DnsName `
lab.contoso.de, *.contoso.de -cert Cert:\LocalMachine\My
```

for creating wildcard certificates.

3.3.2 Extended options in Windows 10 and Server 2016

You can override most of the defaults for new certificates with your own parameters for New-SelfSignedCertificate (bit.ly/37c1Plf), but only from Windows 10 and Server 2016 on. Before that, the cmdlet only accepted the parameters *DnsName*, *CloneCert* und *CertStoreLocation*.

The following command allows you to extend the validity beyond one year by specifying a date:

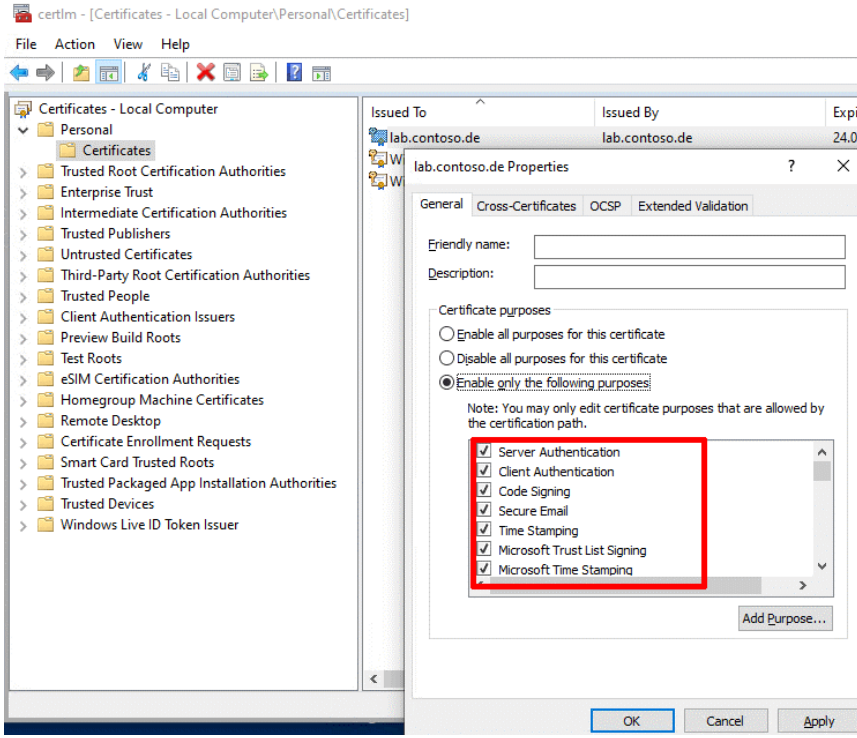
```
New-SelfSignedCertificate -DnsName lab.contoso.de `
-CertStoreLocation Cert:\LocalMachine\My `
-NotAfter (Get-Date).AddYears(2)
```

This example sets the validity to 2 years.

Other use cases besides client and server authentication can also be defined for the certificate. In addition to the default value SSLServerAuthentication, the *Type* parameter also accepts these values:

- CodeSigningCert
- DocumentEncryptionCert
- DocumentEncryptionCertLegacyCsp

On top of that there is *Custom*, which activates all purposes for a certificate. They can be individually deselected again later using the MMC certificate snap-in.



If you select 'Custom' as the type, you generate a certificate with all purposes.

If you do not want the private key to be exportable, you can achieve this using the parameter:

```
-KeyExportPolicy NonExportable
```

3.3.3 Exporting the certificate

If you want to export the certificate to a PFX file in order to use it on an IIS web server, then Export-PfxCertificate serves this purpose. However, it requires that you secure the target file, either with a password or with access rights that you set using the *ProtectTo* parameter.

If you use a password, you first turn it into a secure string:

```
$CertPW = ConvertTo-SecureString -String "secret" `
-Force -AsPlainText
```

It is then passed to the parameter *Password* when calling Export-PfxCertificate:

```
Export-PfxCertificate -Password $CertPW `
-Cert cert:\LocalMachine\My\<Thumbprint> myCert.pfx
```

You specify the certificate via the path in the store and its thumbprint.

Exporting self-signed certificate to a PFX file

If you use a self-signed certificate on a server, it is not considered trustworthy by the clients. To bypass the corresponding warning, you can import it into the trusted root certification authorities on the clients, either manually or via GPO.

To do this, export the certificate without a private key in DER-encoded format:

```
Export-Certificate -FilePath MyCert.cer `
-Cert Cert:\LocalMachine\My\<Thumbprint>
```

In Windows the name extension for such an export file is usually ".cer".

3.4 Remoting over HTTPS with a self-signed certificate

WinRM encrypts data by default and is therefore secure even if you only work with HTTP (which is the standard configuration). Especially in workgroups, you can achieve additional security by using HTTPS, whereby a self-signed certificate should suffice in most cases.

Indeed, Microsoft's documentation for *Invoke-Command* (bit.ly/3fVd2d1) confirms that WS-Management encrypts all transmitted PowerShell data. Unfortunately, if not configured properly, PowerShell Remoting is insecure and ~~it~~ in some cases you need to change the default configuration.

To check how your machines are configured, you can run this command:

```
winrm get winrm/config
```

```
PS C:\Windows\system32> winrm get winrm/config
Config
    MaxEnvelopeSizekb = 500
    MaxTimeoutms = 60000
    MaxBatchItems = 32000
    MaxProviderRequests = 4294967295
    Client
        NetworkDelayms = 5000
        URLPrefix = wsman
        AllowUnencrypted = false
        Auth
            Basic = true
            Digest = true
            Kerberos = true
            Negotiate = true
            Certificate = true
            CredSSP = false
        DefaultPorts
            HTTP = 5985
            HTTPS = 5986
        TrustedHosts
    Service
        RootSDDL = O:NSG:BAD:P(A;;GA;;;BA)(A;;GR;;;IU)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD)
        MaxConcurrentOperations = 4294967295
        MaxConcurrentOperationsPerUser = 1500
        EnumerationTimeoutms = 240000
        MaxConnections = 300
        MaxPacketRetrievalTimeSeconds = 120
        AllowUnencrypted = false
        Auth
            Basic = false
```

Checking WinRM configuration

You can also view the configuration in PowerShell:

```
dir WSMan:\localhost\Service | ? Name -eq AllowUnencrypted
```

```
Administrator: Windows PowerShell
PS C:\Windows\system32> dir WSMan:\localhost\Service | ? Name -eq AllowUnencrypted


   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Service

Type            Name                             SourceOfValue   Value
----            ----                             -------------   -----
System.String   AllowUnencrypted                                 false


PS C:\Windows\system32> dir WSMan:\localhost\Client | ? Name -eq AllowUnencrypted


   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Client

Type            Name                             SourceOfValue   Value
----            ----                             -------------   -----
System.String   AllowUnencrypted                                 false
```

Query current WS-Management configuration using PowerShell

For the client, the corresponding command is

```
dir WSMan:\localhost\Client | ? Name -eq AllowUnencrypted
```

3.4.1 Additional protection for workgroup environments

The second and, in my view, bigger problem is that, if you are working with machines that are not in an Active Directory domain, you don't have any trust relationship with the remote computers. You are then dealing only with symmetric encryption, so man-in-the-middle attacks are theoretically possible because the key has to be transferred first.

There you have to add the remote machines that are not in an Active Directory domain to your TrustedHosts list on the client. However, you don't improve security just by defining IP addresses or computer names as trustworthy. This is just an extra hurdle that Microsoft added so you know that you are about to do something risky.

This is where PowerShell Remoting via SSL comes in. For one, HTTPS traffic is always encrypted. Thus, you can always automate your tasks remotely, free of worry. And, because SSL uses asymmetric encryption and certificates, you can be sure that you are securely and directly connected to your remote machine and not to the computer of an attacker that intercepts and relays your traffic.

On the downside, configuring PowerShell Remoting for use with SSL is a bit more difficult than just running *Enable-PSRemoting*. The main problem is that you need an SSL certificate. If you just want to manage some standalone servers or workstations, you probably don't like to acquire a publicly-signed certificate and want to work with a self-signed certificate instead.

However, you will now see that enabling SSL for WinRM on the client and on the server is not so difficult (although it is not as straightforward as with SSH), and you can do it all with PowerShell's built-in cmdlets. You don't even need the notorious *winrm* Windows command-line tool.

3.4.2 Enabling HTTPS on the remote computer

The first thing we need to do is create an SSL certificate. If you have a publicly-signed certificate, things are easier and you can use

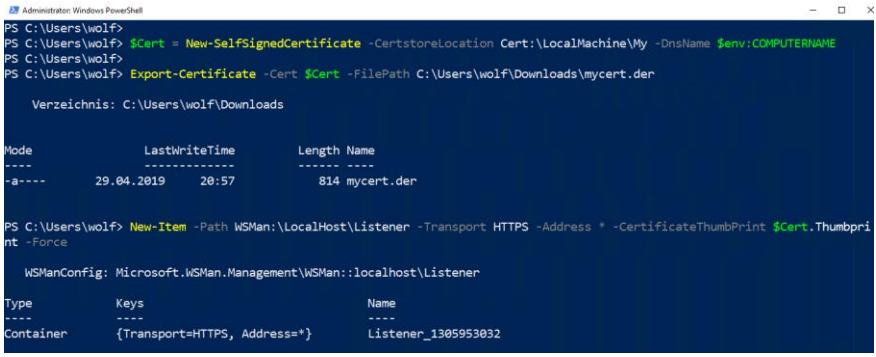
```
Set-WSManQuickConfig -UseSSL
```

As mentioned above, since the release of PowerShell 4, we don't require third-party tools for issuing a self-signed certificate.

The *New-SelfSignedCertificate* cmdlet is all we need:

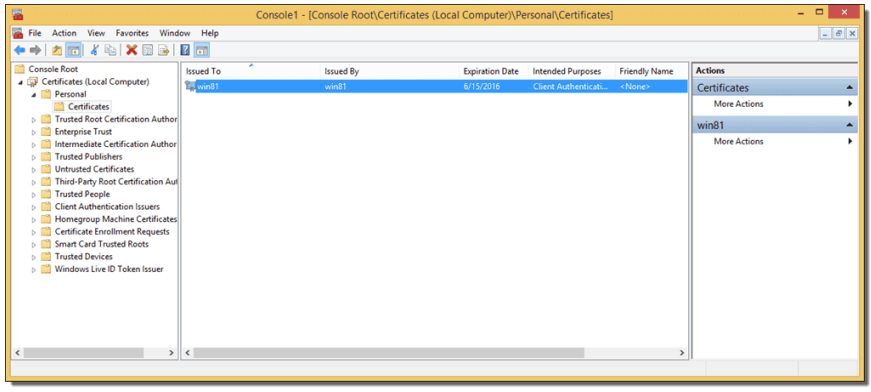
```
$Cert = New-SelfSignedCertificate -DnsName "myHost" `
-CertstoreLocation Cert:\LocalMachine\My
```

It is important to pass the name of the computer that you want to manage remotely to the *-DnsName* parameter. If the computer has a DNS name, you should use the fully qualified domain name (FQDN).



Issue self-signed certificate, export it, and generate HTTPS listener for PowerShell remoting.

If you want to, you can verify that the certificate has been stored correctly using the certificate add-in of the Microsoft Management Console (MMC). Type **mmc** on the Start screen and add the Certificates add-in for a computer account and the local computer. The certificate should be in the Personal\Certificates folder.



Certificate in MMC on the remote computer

We now have to export the certificate to a file because we will have to import it later into our local machine. You can do this with the MMC add-in, but we'll do it in PowerShell:

```
Export-Certificate -Cert $Cert -FilePath C:\temp\cert
```

The file name doesn't matter here.

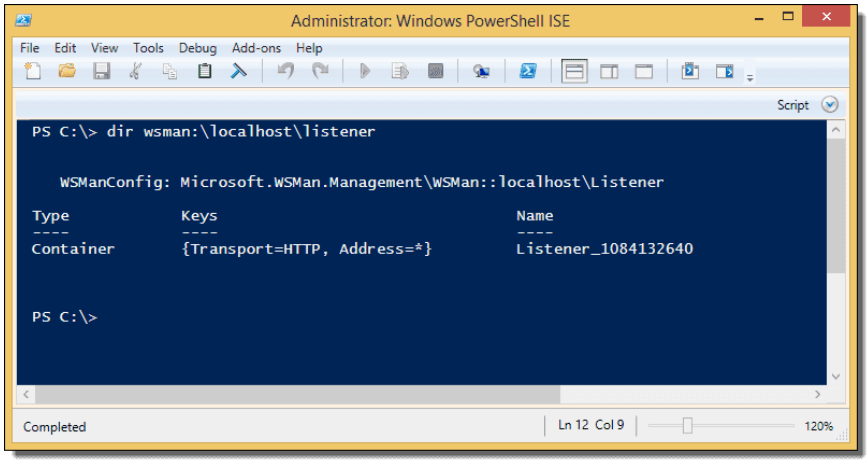
We need the certificate to start the WS-Management HTTPS listener. But we should first enable PowerShell Remoting on the host:

```
Enable-PSRemoting -SkipNetworkProfileCheck -Force
```

The *-SkipNetworkProfileCheck* switch ensures that PowerShell won't complain if your network connection type is set to Public.

Enable-PSRemoting also starts a WS-Management listener, but only for HTTP. If you want to, you can verify this by reading the contents of the WSMan drive:

```
dir wsman:\localhost\listener
```



Listing WSMan listeners

To ensure that nobody uses HTTP to connect to the computer, you can remove the HTTP listener this way:

```
Get-ChildItem WSMan:\Localhost\listener |
Where -Property Keys -eq "Transport=HTTP" |
Remove-Item -Recurse
```

This command removes all WSMan listeners:

```
Remove-Item -Path WSMan:\Localhost\listener\listener* `
-Recurse
```

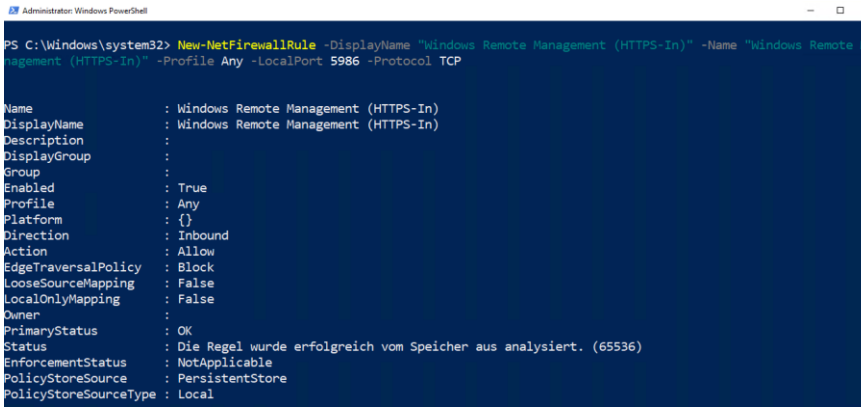
Next, we add our WSMan HTTPS listener:

```
New-Item -Path WSMan:\LocalHost\Listener -Transport HTTPS
-Address * -CertificateThumbPrint $Cert.Thumbprint -Force
```

We are using the $Cert variable that we defined before to read the Thumbprint, which allows the *New-Item* cmdlet to locate the certificate in our certificates store.

The last thing we have to do is configure the firewall on the host because the *Enable-PSRemoting* cmdlet only added rules for HTTP:

```
New-NetFirewallRule -LocalPort 5986 -Protocol TCP `
-DisplayName "Windows Remote Management (HTTPS-In)" `
-Name "Windows Remote Management (HTTPS-In)" -Profile Any
```



Create new firewall rule for PowerShell remoting over HTTPS

Notice here that we allow inbound traffic on port 5986. WinRM 1.1 (current version is 3.0) used the common HTTPS port 443. You can still use this port if the host is behind a gateway firewall that blocks port 5986:

```
Set-Item WSMan:\localhost\Service\EnableCompatibility-
HttpsListener -Value true
```

Of course, you then have to open port 443 in the Windows Firewall. Note that this command won't work if the network connection type on this machine is set to Public. In this case, you have to change the connection type to private:

```
Set-NetConnectionProfile -NetworkCategory Private
```

For security reasons, you might want to disable the firewall rule for HTTP that *Enable-PSRemoting* added:

```
Disable-NetFirewallRule -DisplayName "Windows Remote Management (HTTP-In)"
```

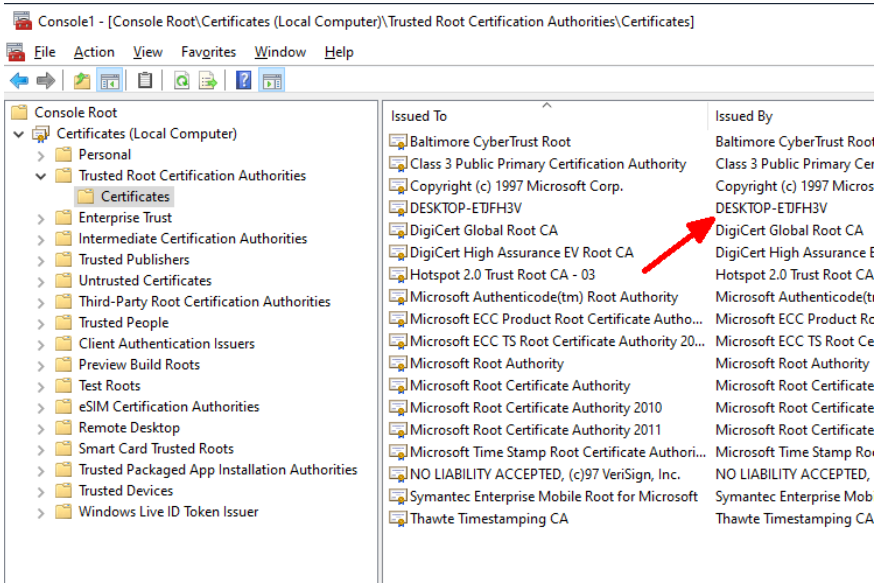
Our remote machine is now ready for PowerShell Remoting via HTTPS, and we can configure our local computer.

3.4.3 Activate HTTPS on the local computer

Things are a bit easier here. First, you have to copy the certificate file to where we exported our certificate. You can then import the certificate with this command:

```
Import-Certificate -Filepath "C:\temp\cert" `
-CertStoreLocation "Cert:\LocalMachine\Root"
```

Note that we need to store the certificate in the Trusted Root Certification Authorities folder here and not in the Personal folder as we did on the remote computer. Your computer trusts all machines that can prove their authenticity with the help of their private keys (stored on the host) and the certificates stored here.



Certificate in MMC on the local computer

By the way, this is why we don't have to add the remote machine to the TrustedHosts list. In contrast to PowerShell Remoting over HTTP, we can be sure that the remote machine is the one it claims to be. This is the main point of using HTTPS instead of HTTP.

We are now ready to enter a PowerShell session on the remote machine via HTTPS:

```
Enter-PSSession -ComputerName myHost `
-UseSSL -Credential (Get-Credential)
```

The crucial parameter here is *-UseSSL*. Of course, we still have to authenticate on the remote machine with an administrator account.

You might receive this error message:

```
The SSL certificate is signed by an unknown certificate
authority.
```

In that case you can just add the the *-SkipCACheck* parameter.

The *Invoke-Command* cmdlet also supports the *-UseSSL* parameter:

```
Invoke-Command -ComputerName myHost -UseSSL `
-ScriptBlock {Get-Process} -Credential (Get-Credential)
```

3.4.4 Conclusion

HTTPS doesn't just add another encryption layer; its main purpose is to verify the authenticity of the remote machine, thereby preventing man-in-the-middle attacks. Thus, you only need HTTPS if you do PowerShell Remoting through an insecure territory. Inside your local network, with trust relationships between Active Directory domain members, WSMan over HTTP is secure enough.

4 Just Enough Administration

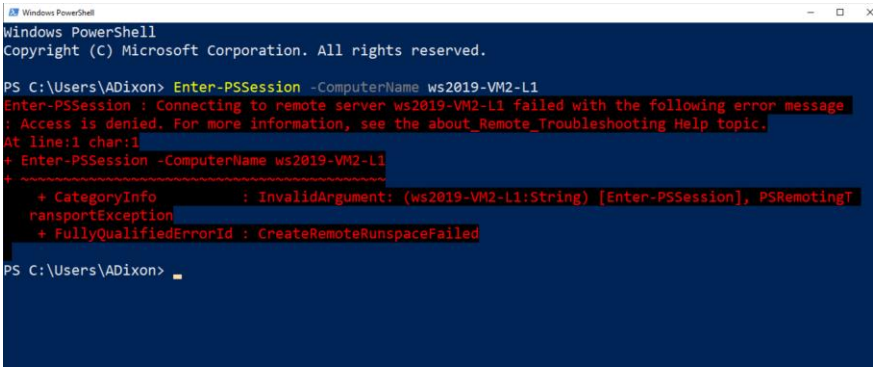
4.1 JEA Session Configuration

If users want to connect to a remote PC via PowerShell without administrative privileges, they fail because of insufficient rights. This limitation can be eliminated with the help of session configurations. Thereby it is not necessary to grant standard users access to all functions of PowerShell.

The ability for remote management is one of the strengths of PowerShell. It is not limited to interactive sessions in which commands are executed on the remote computer. Rather, it also allows you to run scripts to help automate tasks.

4.1.1 Session Configurations as a component of JEA

By default, this option is not available to standard users and their requests will be rejected by the target computer. However, if you want to delegate tasks to employees without administrative privileges, you have to relax this strict rule.



By default, users without administrative rights cannot establish a remote session with PowerShell.

A session configuration serves this purpose. It determines who is allowed to establish a session on a computer. This function is also performed by the Just Enough Administration (JEA). JEA defines what the users are allowed to do there via additional role capabilities files.

In many cases, however, you do not have to deal with the complete JEA, but you can define the access rights and the available language elements directly via a session configuration.

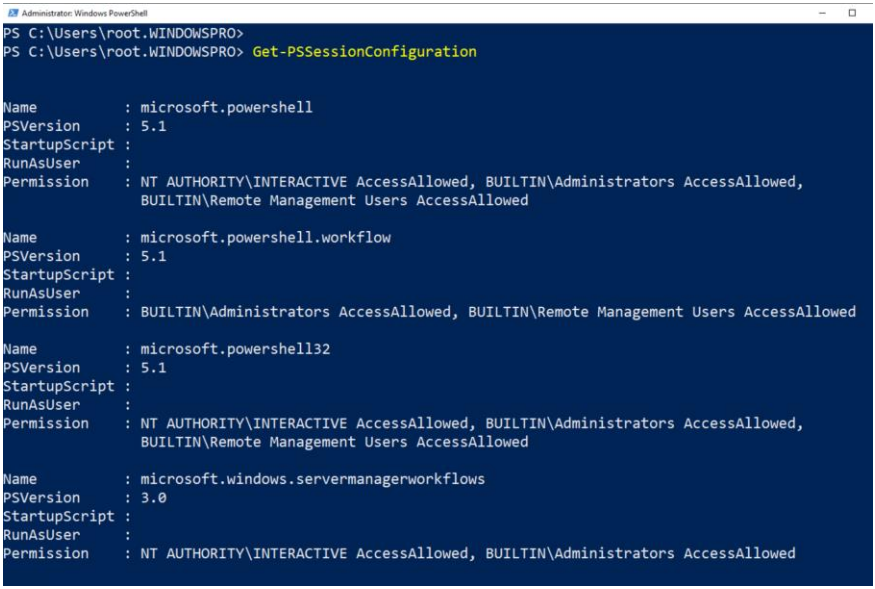
4.1.2 Restrictive standard configurations

Session definitions always control PowerShell access to a computer, even if you have not created one of your own. By default, there are three Session Configurations on each Windows computer, namely *microsoft.powershell*, *microsoft.powershell.workflow* and *microsoft.windows.servermanagerworkflows*.

If you create a new session, such as with *Enter-PSSession*, and do not specify a particular configuration, then *microsoft.powershell* takes effect by default. As you can see from the command

```
Get-PSSessionConfiguration
```

on the target computer, this session configuration is reserved for administrators and members of the local group *Remote Administration Users*.



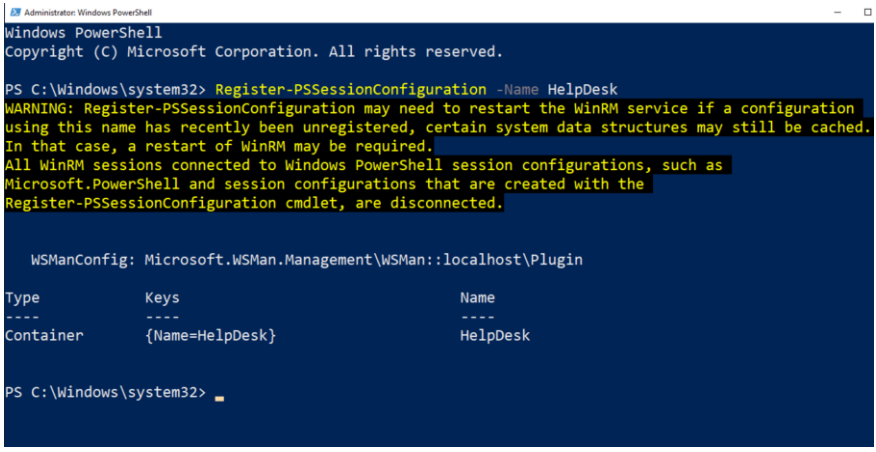
Displaying the existing session configurations and their authorizations with Get-PSSession-Configuration

4.1.3 Defining your own configurations

Theoretically, you could now simply change the security settings of this configuration to give access for selected standard users. But you should refrain from that and maintain a working configuration for admins.

The simplest way to create a new session configuration is to execute a command according to the following pattern on the target computer (also called an endpoint in JEA jargon):

```
Register-PSSessionConfiguration -Name HelpDesk
```



Create a new session configuration with Register-PSSessionConfiguration

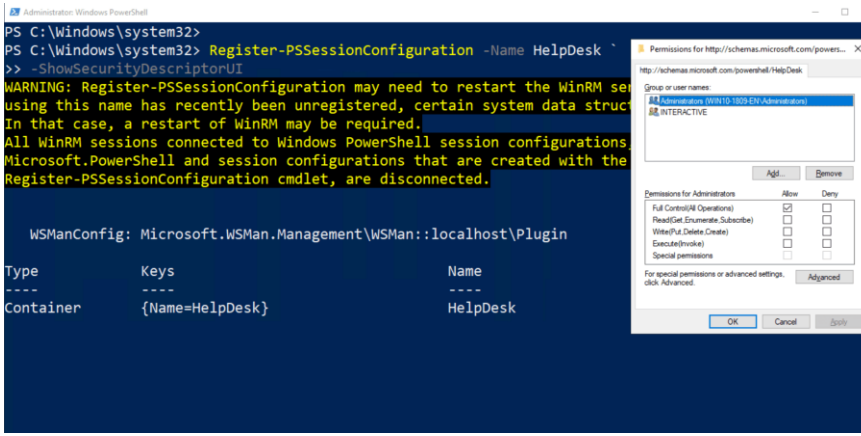
Not much is gained with this command, because the new configuration is only a copy of *microsoft.powershell* and does not allow users other than admins to access the computer. Hence, you should define the permissions when you create the configuration.

4.1.4 Defining permissions

This is done using the parameter *SecurityDescriptorSddl*, but it needs the permissions in the syntax of the Security Descriptor Definition Language (bit.ly/33Xti8f). If you do not need to create Session Configurations too often, you can save yourself this effort and use the parameter *ShowSecurityDescriptorUI* instead:

```
Register-PSSessionConfiguration -Name HelpDesk `
-ShowSecurityDescriptorUI
```

This opens the dialog you already know from managing file permissions.



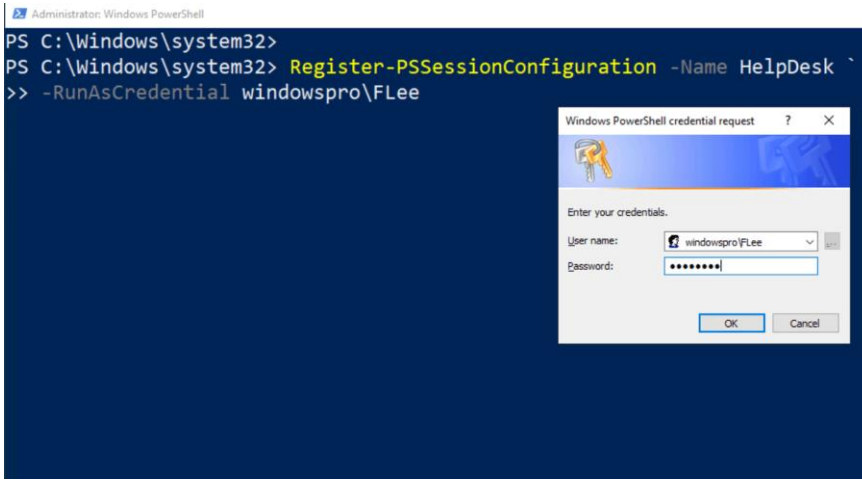
Managing Permissions for a Session Configuration

By adding local or AD groups and assigning them the desired privileges, you determine who can use this configuration. To run a remote session, the *Execute* permission is sufficient here.

4.1.5 Defining RunsAs users

So far you have already configured who is allowed to start a session on this remote computer using the new configuration. In addition, you can also specify under which user ID this should happen by passing the respective ID to the *RunAsCredential* parameter:

```
Register-PSSessionConfiguration -Name HelpDesk `
-RunAsCredential contoso\FLee
```



Specify the account under which the remote session should run if it was started from session configuration.

PowerShell then prompts for the password and stores it in the configuration. If a user then connects to the target PC via a session configuration, he or she will automatically work there in the context of this account. If you do not use this option, the connection is made under the locally logged on user.

4.1.6 Forcing restrictions for sessions

Working under a different account might give the users different permissions in the file system, but functional restrictions imposed by a session configuration apply regardless of the account used. The RunsAs account therefore does not require any permissions in the Security Descriptor of the session configuration.

The *Register-PSSessionConfiguration* cmdlet provides several parameters that can be used to limit the users' options:

- MaximumReceivedDataSizePerCommandMB: specifies the maximum amount of data in MB that can be transferred with one command (Default: 50MB).
- MaximumReceivedObjectSizeMB: determines the maximum size of a single object that can be transferred (Default: 10MB)
- SessionType: decides which modules and snap-ins are available in the session. These are none when the value is *empty* (and must be explicitly added using the *ModulesToImport* parameter, for example). *Default* allows users to extend the functionality themselves using *Import-Module*. Finally, *RestrictedRemoteServer* provides half a dozen cmdlets.

All of the parameters described here, except *Name*, can also be used later to customize the configuration using *Set-SessionConfiguration* (bit.ly/33Y3KIi).

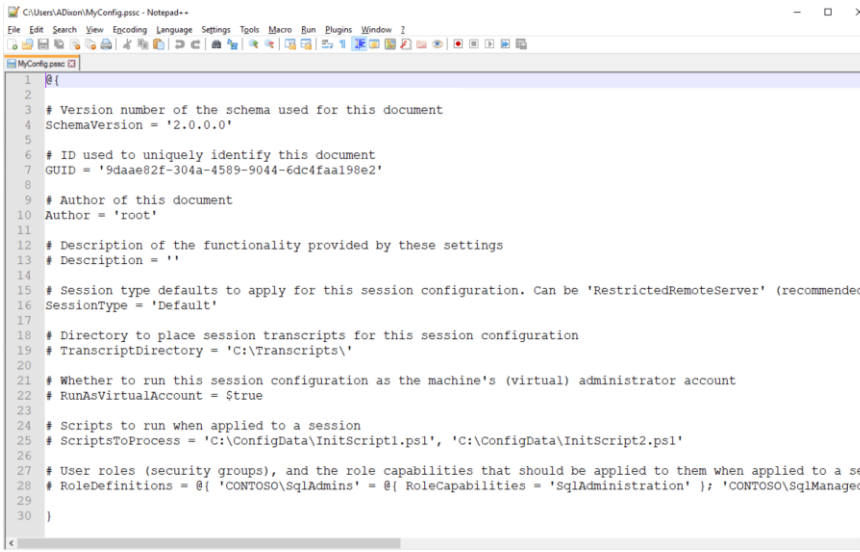
4.1.7 Additional options via configuration file

In many cases, *Register-PSSessionConfiguration* can create the necessary context for users to perform specific tasks on the remote host. As an additional option you can run a script when starting the session (*StartupScript* parameter).

But if that is not enough, there are more options available with a configuration file. This can be created with the *New-PSSessionConfigurationFile* cmdlet. You can pass the desired settings to the configuration file either as parameters (see the complete list here: bit.ly/33Tfeg8) or you can run it in this minimalist form:

```
New-PSSessionConfigurationFile -Path .\MyConfig.pssc
```

The file name requires the .pssc extension. Then open the file in a text editor and add the desired settings, some of which are already available and commented out.



Default file created by New-PSSessionConfigurationFile

The following are particularly useful to prevent users from potentially harmful actions:

- LanguageMode with the values FullLanguage, RestrictedLanguage, ConstrainedLanguage, NoLanguage: The latter allows only the execution of cmdlets and functions, other language resources are not available. FullLanguage offers the full range of language capabilities, the other two lie between these two poles.
- VisibleAliases, VisibleCmdlets, VisibleFunctions, VisibleProviders: These allow you to specify which aliases, cmdlets, functions, and providers are available in the session. You can use wildcards and specify multiple values as array.

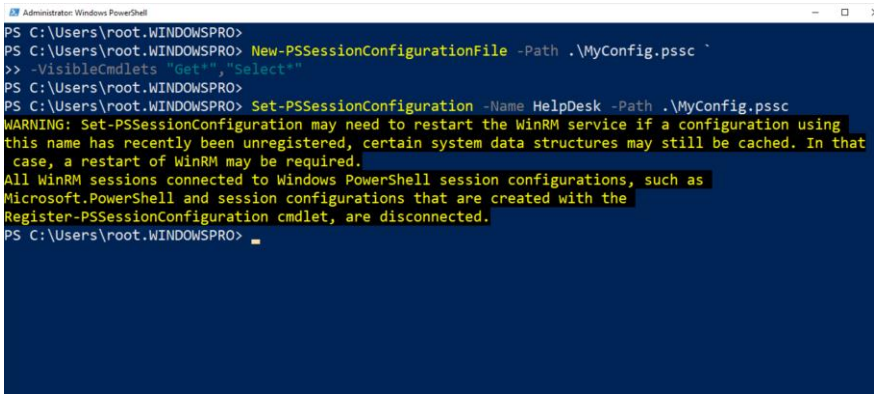
4.1.8 Limiting access to cmdlets

To restrict the available cmdlets to those which only read and do not write, you could use the expression *Get**, *Select**:

```
New-PSSessionConfigurationFile -Path .\MyConfig.pssc `
-VisibleCmdlets "Get*","Select*"
```

Then, you adjust the Session Configuration based on this file:

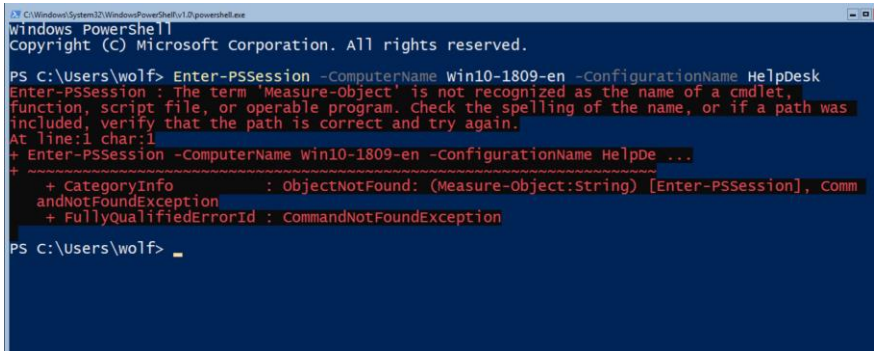
```
Set-PSSessionConfiguration -Name HelpDesk `
-Path .\MyConfig.pssc
```



Create the configuration file and assign it to a new session configuration.

If you now try to establish an interactive remote session with the computer, you will fail, because not all necessary commands are available:

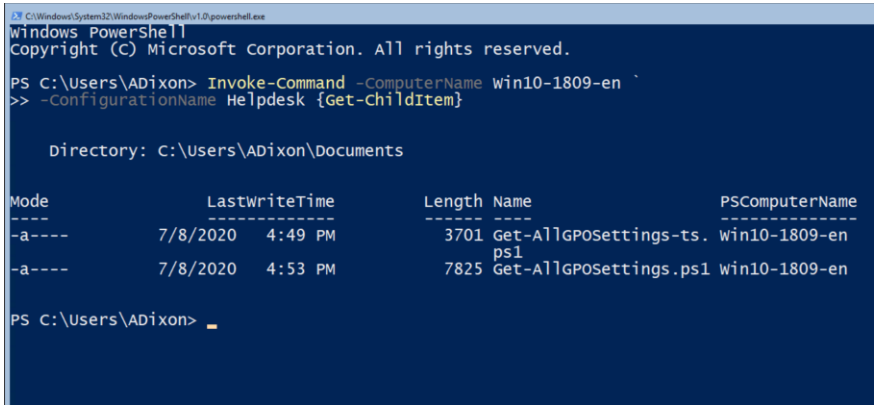
```
Enter-PSSession -ComputerName remote-pc `
-ConfigurationName HelpDesk
```



The reduced range of functions is not sufficient for an interactive session.

Therefore, a user with this session configuration is restricted to issuing commands remotely, for example, using a command like this:

```
Invoke-Command -ComputerName remote-pc `
-ConfigurationName Helpdesk {Get-ChildItem}
```



Issuing a command remotely in a restricted session using Invoke-Command

4.1.9 Assign a configuration to a session

As the two commands above show, you have to specify the desired session configuration using the *ConfigurationName* parameter. If you don't do that, *microsoft.powershell* will be applied and non-administrative users will be kept out. But you can specify which configuration is used by default with the variable *$PSSessionConfigurationName*.

Finally, you can remove session configurations that you no longer need by using the *Unregister-PSSessionConfiguration* cmdlet. It requires only the name of the configuration as its arguments.

4.2 Defining and assigning role functions

Just Enough Administration (JEA) allows users without administrative privileges to perform management tasks. JEA is based on session configurations that determine who gets access. Role capabilities then define the means available for them in PowerShell.

You can already control some of the properties when you create or change a session configuration with *Register-PSSessionConfiguration or Set-PSSessionConfiguration*. You get more options by using a configuration file (.pssc). Here, a certain language mode can be enforced or access to specific cmdlets can be restricted.

However, if you need a more complex set of rules to tailor the options in a session to the needs of specific tasks, then you should define the role functions in a separate .psrc file.

4.2.1 More flexibility using role capability files

This has at least two advantages. First, you have to update a session configuration every time you change role functions directly in its configuration file, and then restart WinRM. In contrast, external role definitions are simply read in at runtime.

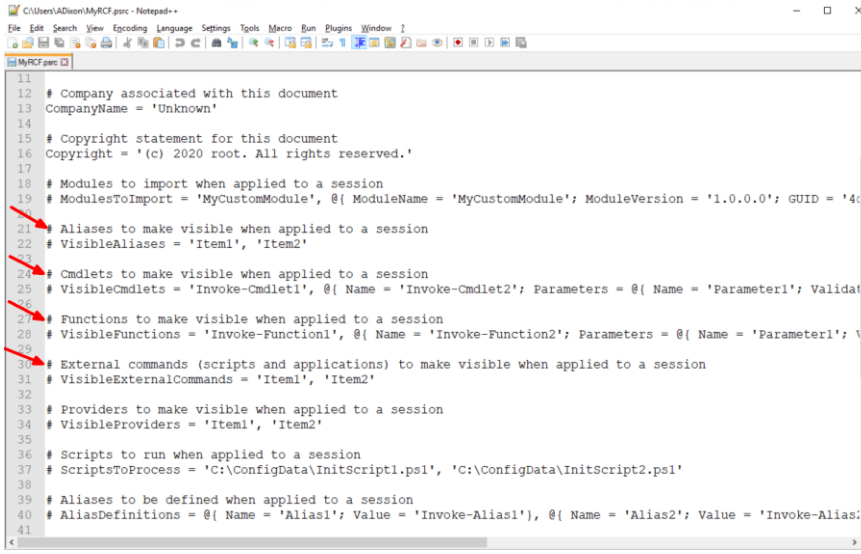
Secondly, independent role capability files can be assigned to several session configurations, so that redundant information can be avoided. Conversely, it is also possible to use several of these role functions in a single session configuration so that they can be structured modularly.

4.2.2 Generating a role capability file

The files with the .psrc extension to describe role capabilities are text files. A skeleton file can be created with the command:

```
New-PSRoleCapabilityFile -Path MyRCF.psrc
```

It contains all available options plus the corresponding description in a commented form, so that you can edit them right away in an editor. When creating the file, you could also use the numerous parameters of *New-PSRoleCapabilityFile* (bit.ly/2NTpO12) to set various settings.

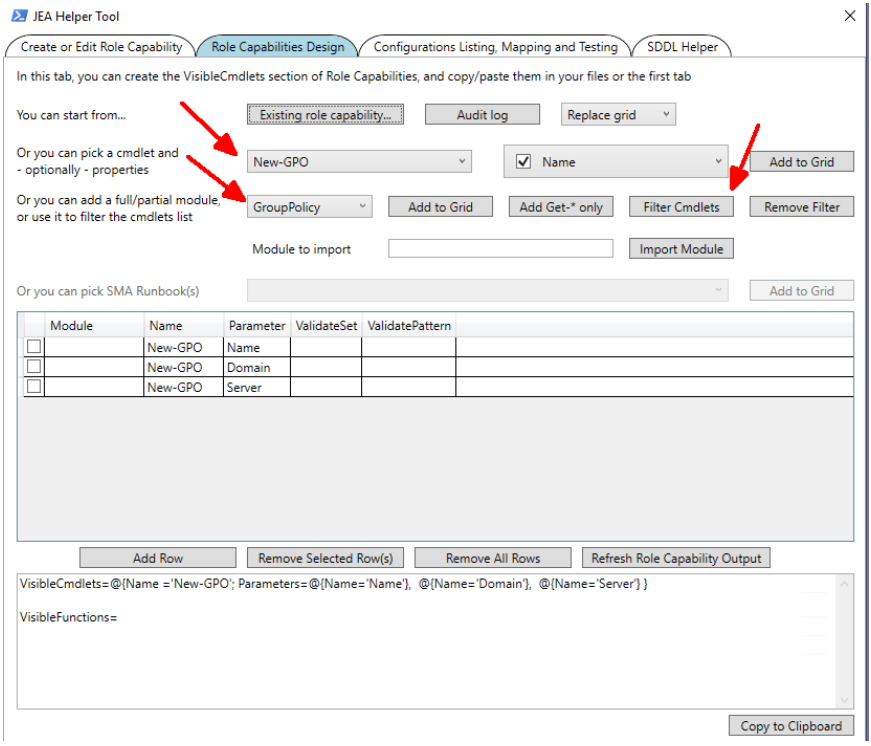


Example of a Role Capability File and its options

One of the most important aspects of a role definition is to restrict sessions to specific cmdlets, functions, aliases, or variables. The use of cmdlets can be limited down to the level of individual parameters.

4.2.3 Compiling VisibleCmdlets via GUI

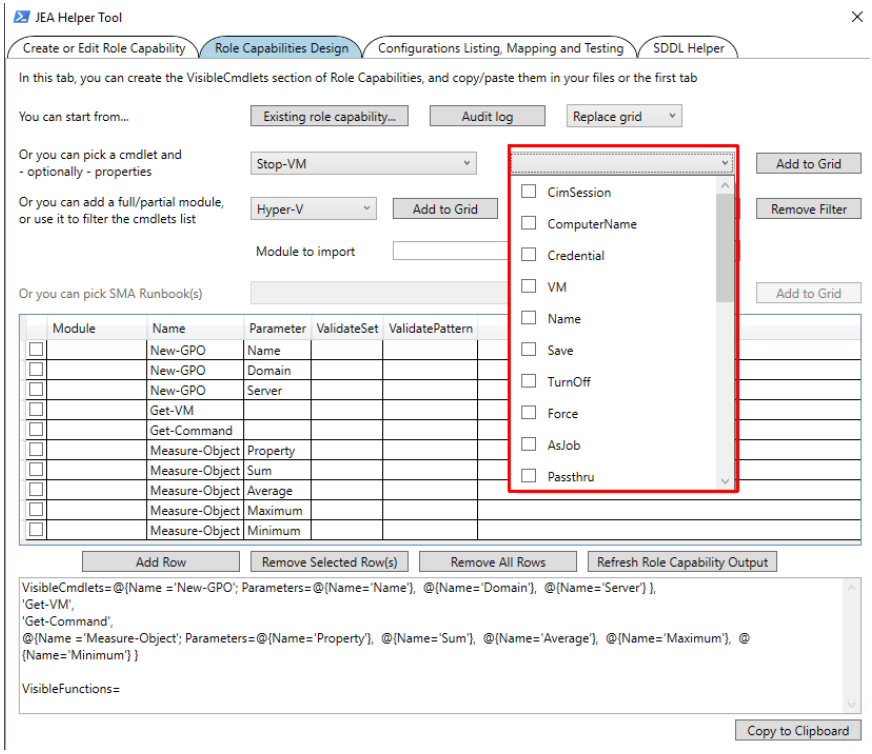
It is relatively time-consuming if you want to manually enter such detailed information in the .psrc file. This job is simplified by the JEA Helper Tool (bit.ly/2OlLUYX), a PowerShell script with GUI. On the *Role Capabilities Design* tab, you can interactively compile the list of cmdlets that the users of a particular session are allowed to see.



Selecting the cmdlets that you want to use for a session configuration

If you select a module from the drop-down in the third row and then click on *Filter Cmdlets*, the list in the second row is reduced to the cmdlets of that module. After you have selected a cmdlet, a drop-down menu opens

next to it with all of its parameters. Here you can select individual parameters or mark none of them in order to enable all of them.



Selecting the allowed parameters of a cmdlet

The tool offers additional features such as creating a .psrc skeleton with *New-PSRoleCapabilityFile* or a new session configuration. Because of the cumbersome operation, you will usually do without it.

4.2.4 Saving the role capability file

Once you have created the list of permitted cmdlets and parameters, you can add them to the .psrc file. You save this file in a directory called *RoleCapabilities* under

```
$env:ProgramFiles\WindowsPowerShell\Modules
```

4.2.5 Assigning role functions to session configuration

The last step is to link the role capabilities to the desired session configuration. To do this, edit the configuration file with the extension .pssc and add the role functions there.

Since you create this file automatically at the beginning, this (commented out) section for *RoleDefinitions* should already be there:

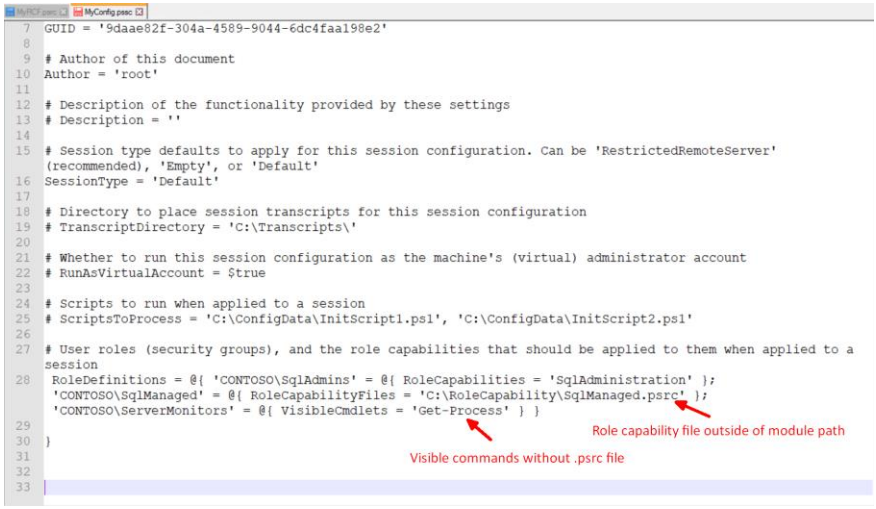
```
# RoleDefinitions = @{ 'CONTOSO\SqlAdmins' = `
@{ RoleCapabilities = 'SqlAdministration' };

'CONTOSO\SqlManaged' = @{ RoleCapabilityFiles =
'C:\RoleCapability\SqlManaged.psrc' };

'CONTOSO\ServerMonitors' = `
@{ VisibleCmdlets = 'Get-Process' } }
```

Following the same pattern, you now add your own entry, whereby you have 3 options, as shown in the example. The last of these defines the allowed cmdlets directly in the Session Configuration File and is therefore not applicable if you use a .psrc file.

If you save your .psrc file under the name *SqlManaged.psrc* in the module path as described above, the entry could look like this:

```
RoleDefinitions = @{ 'contoso\SqlAdmins' = `
@{ RoleCapabilities = 'SqlAdministration' }};
```

This gives the *SqlAdmins* group from the contoso domain the role capabilities defined in *SqlManaged.psrc*.



Options for defining role capabilities in a session configuration file

If you have chosen a different location to save the file, then you have to proceed as shown in the last entry in the example and enter the name of the file including the path as value for *RoleCapabilityFiles*.

Finally, you have to update the session configuration using the following command:

```
Set-PSSessionConfiguration -Name MySessionConfig `
-Path .\MyConfig.pssc
```

5 Audit PowerShell activities

5.1 Log commands in a transcription file

In order to detect the abuse of PowerShell, you can record all executed commands and scripts. There are two mechanisms for this, one of them writes all input and output to a file. It is recommended to store the collected data in a central location.

Microsoft describes the form of recording, where PowerShell logs all processed inputs and the resulting output in one file, as "over-the-shoulder-transcription". This term reflects that PowerShell writes to a file what an observer would see when he looks over the shoulder of the user during his PowerShell session.

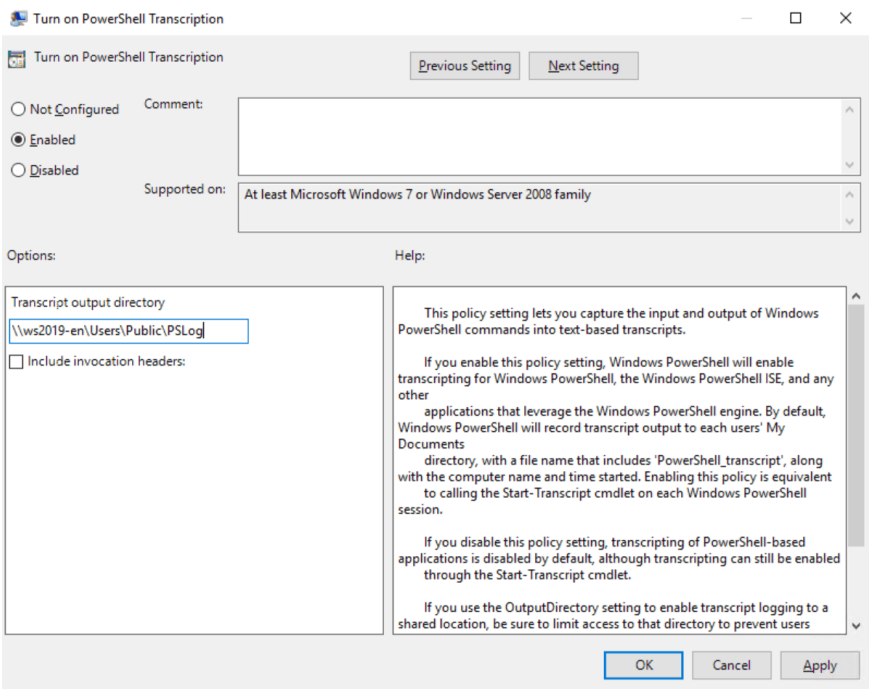
5.1.1 Activate logging using a cmdlet

This variant has been around since the early days of PowerShell and, in the past, could be controlled only explicitly by using the *Start-Transcript* and *Stop-Transcript* cmdlets. To enable automatic recording of the commands, you had to include the *Start-Transcript* call in the PowerShell profile.

Not only is this cumbersome if you have to configure many machines in this way, but it is also relatively easy for an attacker to circumvent this method. However, explicitly starting and stopping the recording using a cmdlet can be useful if you include it in your own scripts to see what output they produce.

5.1.2 Enabling transcripts via GPO

Since PowerShell 5, you can turn on transcripts using group policy. The corresponding setting is called *Turn on PowerShell Transcription* and can be found under *Policies => Administrative Templates => Windows Components => Windows PowerShell*.

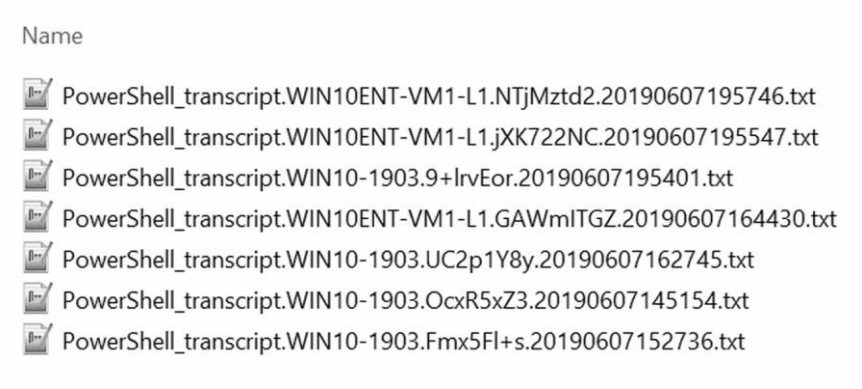


Enable PowerShell transcripts via GPO. Optionally, specify a separate directory and activate the timestamp.

If you activate it under both branches (computer and user configuration), the setting is enforced at the computer level.

5.1.3 Own log file for each session

By default, the feature creates a directory in the user's profile for each day and writes the entries for each session to a separate text file, whose name consists of "PowerShell_transcript" plus the hostname of the computer and a random number.



PowerShell creates a separate log file for each session on each computer.

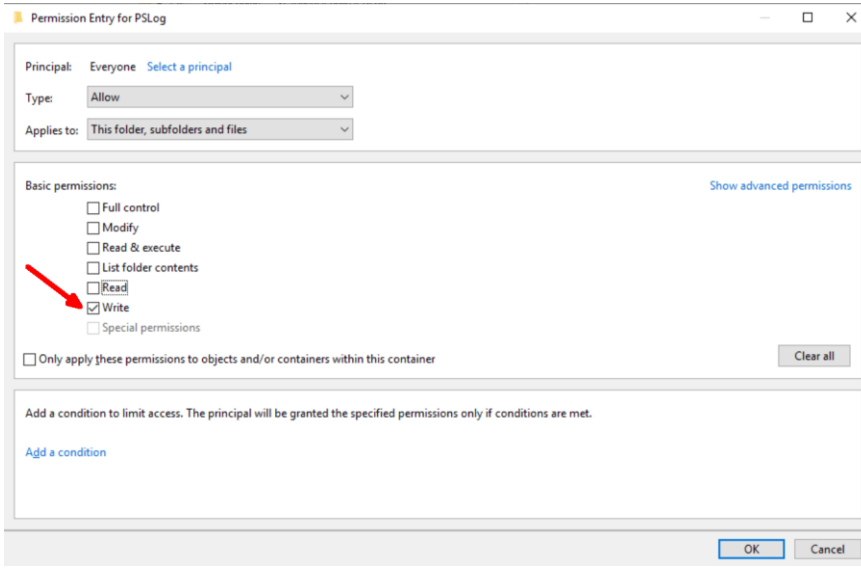
Of course, it makes sense to store the records centrally on a shared directory on the network. The *Start-Transcript cmdlet* uses the *OutputDirectory* parameter to redirect output from the default directory to another. The GPO setting for activating the transcripts includes a separate input field for this purpose.

5.1.4 Protecting the log directory

Usually you will want to avoid that users read or even change the contents of these log files. On the one hand, they may contain sensitive information such as passwords, on the other hand, the necessary write permission

would make it easy for an attacker to cover his tracks. Therefore, you have to prevent users from viewing the files and their contents.

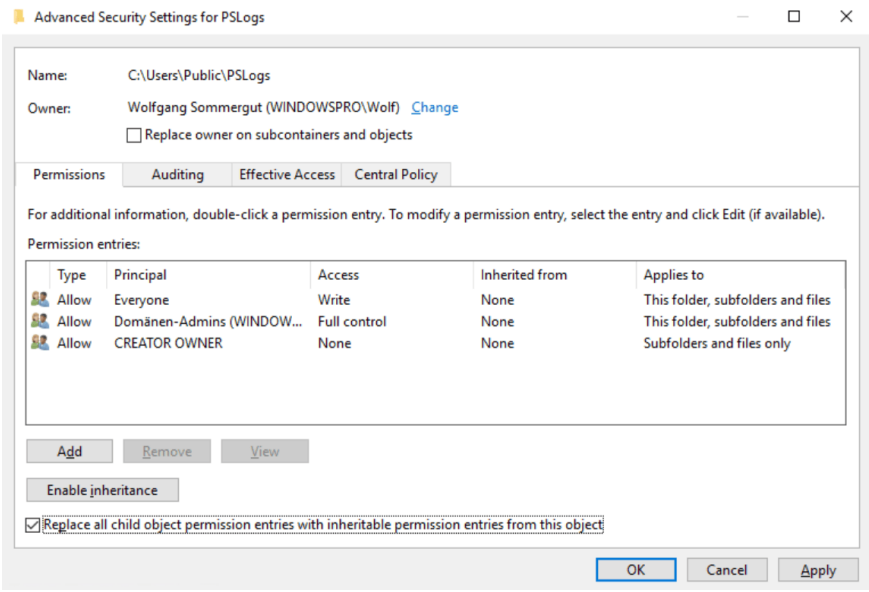
For this purpose, Microsoft recommends restricting the NTFS rights on the shared directory.



Everybody' gets only the rights to 'Read' and 'Write'.

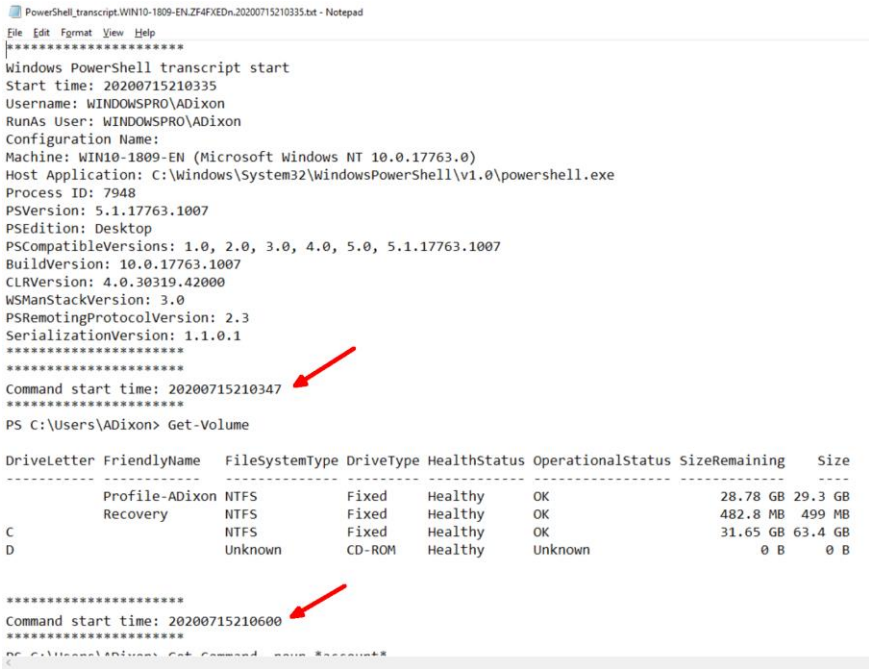
Specifically, you should proceed as follows:

- Disable inheritance for the configured log directory, remove all existing permissions
- Administrators get full access
- *Everyone* gets the right to '*Write*'
- Creator owner is deprived of all rights



Permissions for the PowerShell log directory

Another option for both *Start-Transcript* and GPO settings is to write a header for each call. This contains a timestamp for the respective command.



Transcript with header and timestamp for each command

If this option is used, the volume of recorded data increases considerably. Since the header in each file already contains detailed information about the session, you will usually not need the additional time stamp for each action.

5.1.5 GPO does not work for PowerShell 6/7

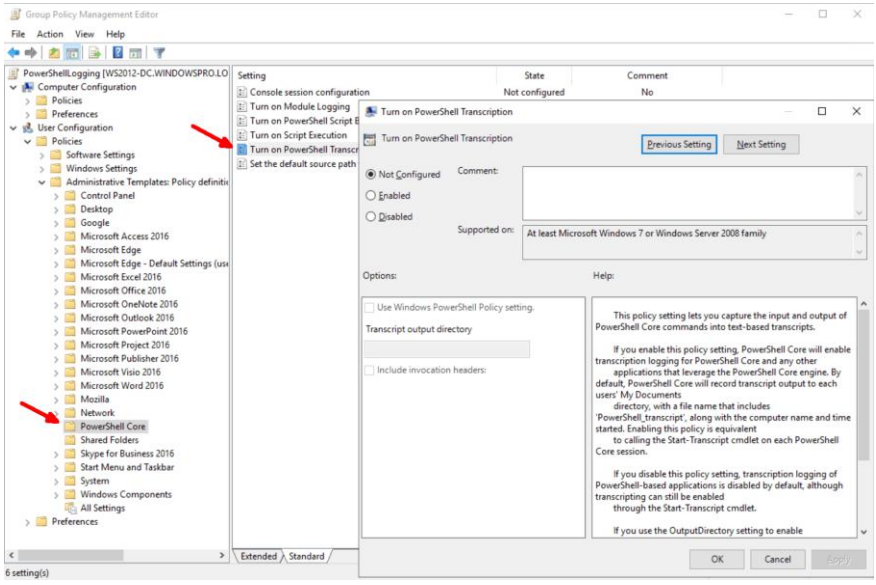
The *PowerShellExecutionPolicy.admx* administrative template writes only the registry values for Windows PowerShell, so that *EnableTranscripting* does not affect PowerShell Core or PowerShell 7.

For version 6, you must therefore set the required key in the registry yourself. The following content for a .reg file shows the names of the two DWORDs and the path where you have to create them.

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Pow-
erShellCore\Transcription]
"EnableTranscripting"=dword:00000001
"OutputDirectory"="\\server\\pslogs"
```

If you want to set these settings on a larger number of computers, it is recommended to adjust the registry using the Group Policy Preferences.

PowerShell 7 comes with its own ADMX template which can be copied to %systemroot%\policydefinitions or to the Central Store. The settings for version 7 are located in the GPO Editor directly under *Administrative Templates* in the *PowerShell Core* container (both computer and user).



Enabling Transcription Logging for PowerShell 7 via Group Policy

The policies are largely identical to those for Windows PowerShell, and the same is true for *Turn on PowerShell Transcription*. It is particularly useful that each setting has the option *Use Windows PowerShell Policy setting* so that you don't have to manage PowerShell 7 separately.

5.2 Scriptblock logging: Record commands in the event log

To detect suspicious activities, it is helpful to have all executed commands recorded. In addition to recording the history in a text file, PowerShell has also supported logging in the event log since version 5.

PowerShell v5 included several innovations in logging. It extended the older method, the so-called "over-the-shoulder transcription," to all PS hosts, including ISE, and hence was no longer limited to the command line. Furthermore, this feature can now also be activated via group policies.

5.2.1 Logging the actual commands

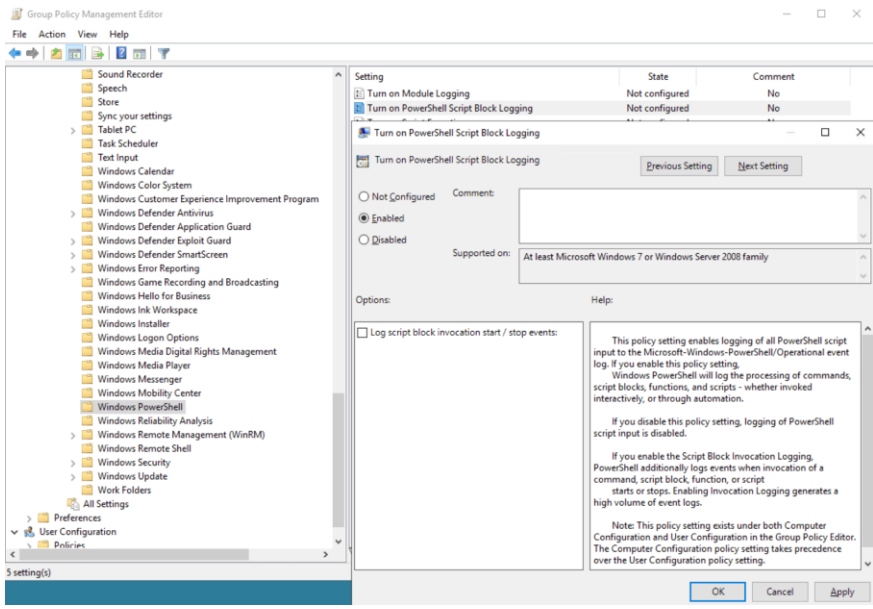
The recording of all commands in a text file has been complemented by the so-called *deep scriptblock logging*. It not only uses the Windows event log instead of a text file, but also records all commands exactly as executed by PowerShell. This way, malicious activity does not easily go unnoticed.

This applies, for example, to the use of dynamic code generation, where commands are stored in a variable and then executed with the help of *Invoke-Expression*. The feature also reveals attempts to hide command sequences by encoding them using Base64.

5.2.2 Activation only via GPO

While transcriptions can also be explicitly turned on and off using the *Start-Transcript and Stop-Transcript* cmdlets, you can enable script block

logging only by using GPOs or by setting the appropriate registry key directly. Therefore, there is still a need for the older method, such as recording the output in your own scripts.



Group policy to enable deep scriptblock logging

The relevant GPO setting is called *Turn on PowerShell Script Block Logging* and can be found under *Policies > Administrative Templates > Windows Components > Windows PowerShell*. If you configure it under Computer and User Configuration, the former setting prevails.

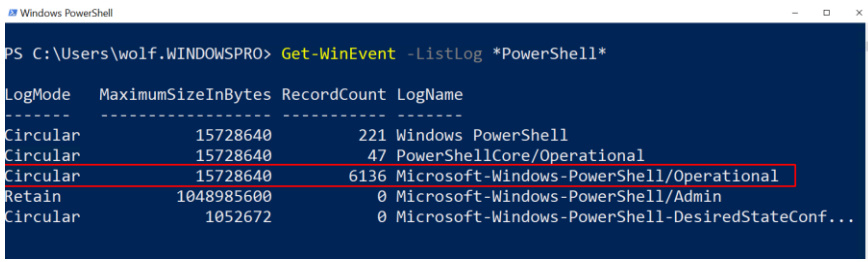
If you select the option for *start/stop*, then you should expect a considerably higher data volume because markers for the start and stop of all events will be written to the log.

5.2.3 Preparing the event log

While you prepare the logging in text files by creating a directory on a file share and assigning the necessary access rights, different preparatory work is required for the newer logging.

Start by changing the maximum size of the event log from the default of 20 MB to a significantly higher value. This is required for two reasons: First, depending on the configuration of the logging feature, a relatively large amount of data is accumulated. Second, attackers should not be able to simply cover their tracks by filling up the log relatively quickly with unsuspicious entries.

Since the evaluation of the logs is left either to scripts developed for this purpose or to SIEM tools, the recorded events are needed at a central location. For this purpose, forward the entries written by PowerShell to a computer in the network.

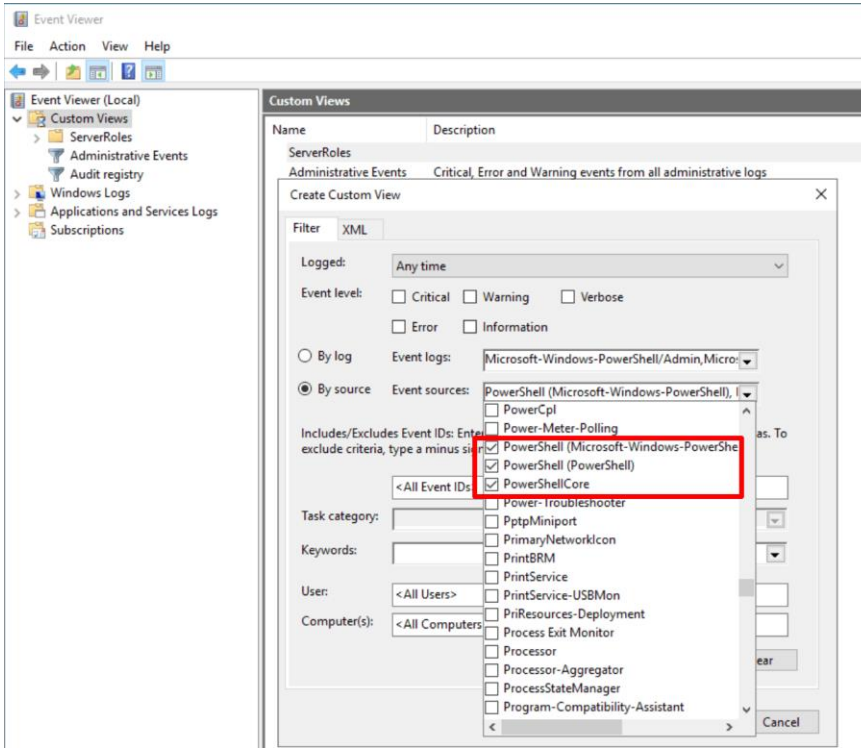


The logging is done under PowerShell/Operational

5.2.4 Event IDs

The logging takes place in the application log under *Microsoft=> Windows => PowerShell => Operational*, and the commands are recorded under

event ID 4104. If you also record start and stop events, these appear under the IDs 4105 and 4106.



Custom filter in the event viewer for recorded script blocks

If you want to set up a user-defined filter for the recorded commands in the event viewer, activate as source

- PowerShell (Microsoft-Windows-PowerShell),
- PowerShell (PowerShell)
- PowerShellCore

In addition, select *Warning* as the event type and enter 4104 as the ID.

5.2.5 Merging command sequences

While transcripts can write their data to a text file with virtually no limits, the script block field in the event log limits the length of the record. Therefore, longer scripts are split up and span several entries.

On Microsoft Docs, there is a template for a PowerShell script that can be used to reassemble the log fragments. If for example you want to string together all recordings for a process with ID 6524, then you could proceed as follows:

```
$created = Get-WinEvent -FilterHashtable `
@{ProviderName="Microsoft-Windows-PowerShell"; Id=4104} |
where ProcessId -eq 6524
$sortedScripts = $created | sort {$_.Properties[0].Value}
$mergedScript = -join ($sortedScripts |
foreach {$_.Properties[2].Value})
```

5.2.6 Script block logging for PowerShell Core

As with transcripts, group policy enables logging of script blocks only for Windows PowerShell. It has no effect on PowerShell Core 6.x and its successor, PowerShell 7.

If you want to record the commands for version 6.x in the event log, you have to set the registry key yourself. To do this, create the *ScriptBlockLogging* key under

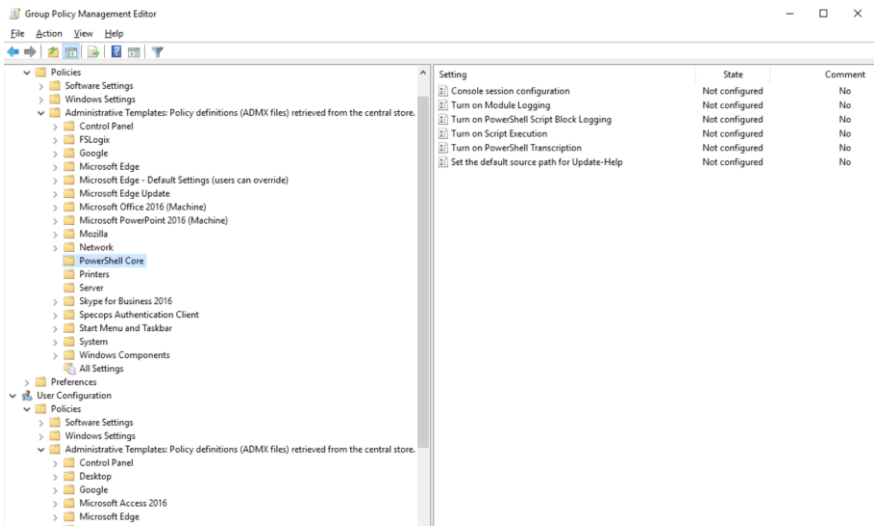
HKLM\SOFTWARE\Policies\Microsoft\PowerShellCore

and assign the value 1 to *EnableScriptBlockLogging*.

The following instructions in a .reg file will accomplish this task:

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Pow-
erShellCore\ScriptBlockLogging] "EnableScriptBlockLog-
ging"=dword:00000001
```

PowerShell 7, on the other hand, includes its own ADMX template, which you can copy to %systemroot%\policydefinitions or to the central store. It contains all the settings known from PowerShell 5, including those for scriptblock logging.



Group policy settings for PowerShell 7

Finally, it should be noted that the log entries for PowerShell Core are located directly under the *Applications and Services* logs. The event IDs for logging are the same as for Windows PowerShell.

5.3 Issuing certificates for document encryption

Beginning with version 5, PowerShell supports the IETF standard Cryptographic Message Syntax (CMS) to encrypt data or log entries. It requires a certificate that has been issued specifically for this purpose. If you want to request the certificate from a Windows CA, you must first set up a template for it.

Microsoft's instructions, for example, for Protect-CmsMessage (bit.ly/2XPVQzB), always describe the procedure for issuing a self-signed certificate with certreq.exe for document encryption. They pack the data for requesting the certificate into an .inf file according to the following pattern:

```
[Version]
Signature = "$Windows NT$"
[Strings]
szOID_ENHANCED_KEY_USAGE = "2.5.29.37"
szOID_DOCUMENT_ENCRYPTION = "1.3.6.1.4.1.311.80.1"
[NewRequest]
Subject = "cn=me@somewhere.com"
MachineKeySet = false
KeyLength = 2048
KeySpec = AT_KEYEXCHANGE
HashAlgorithm = Sha1
Exportable = true
RequestType = Cert
```

```
KeyUsage = "CERT_KEY_ENCIPHERMENT_KEY_USAGE |
CERT_DATA_ENCIPHERMENT_KEY_USAGE"
ValidityPeriod = "Years"
ValidityPeriodUnits = "1000"
[Extensions]
%szOID_ENHANCED_KEY_USAGE% = "{text}%szOID_DOCUMENT_EN-
CRYPTION%"
```

To request the certificate, use the command:

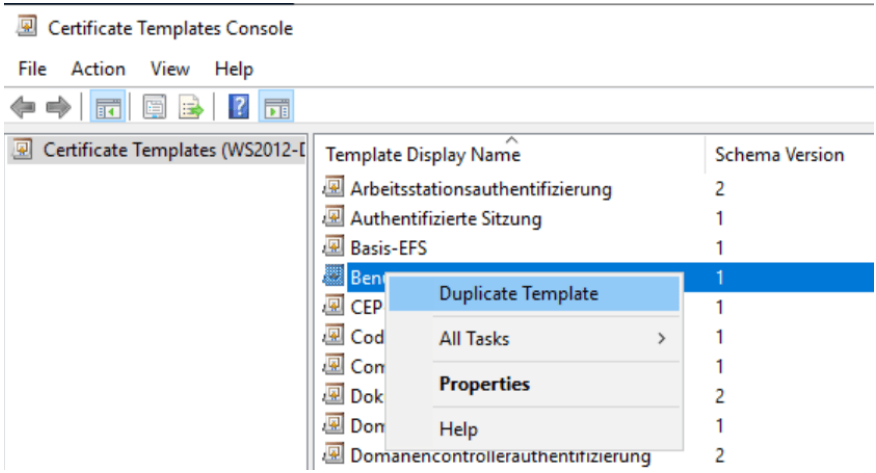
```
certreq -new <INF-file-name>.inf <Certificate-name>.cer
```

The certificate is automatically copied to the local certificate store of the logged-on user. If needed, you can export it and transfer it via GPO to the computers on which you want to encrypt data (bit.ly/30OQJ4o).

5.3.1 Creating a template for enterprise CA

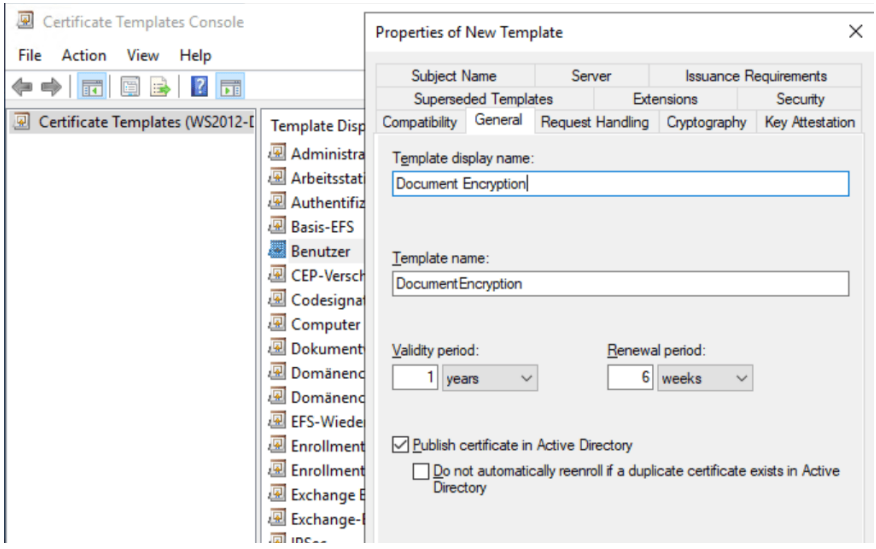
If you prefer a certificate issued by an internal Windows CA instead of a self-signed certificate, the required template is missing by default. If you want to create one, you can follow the settings of the above .inf file.

First, open the Certificate Templates Console, *certtmpl.msc*, and duplicate a suitable existing template. In our example, we will use the template *User*.



Duplicate an existing template as a basis for the new template for document encryption

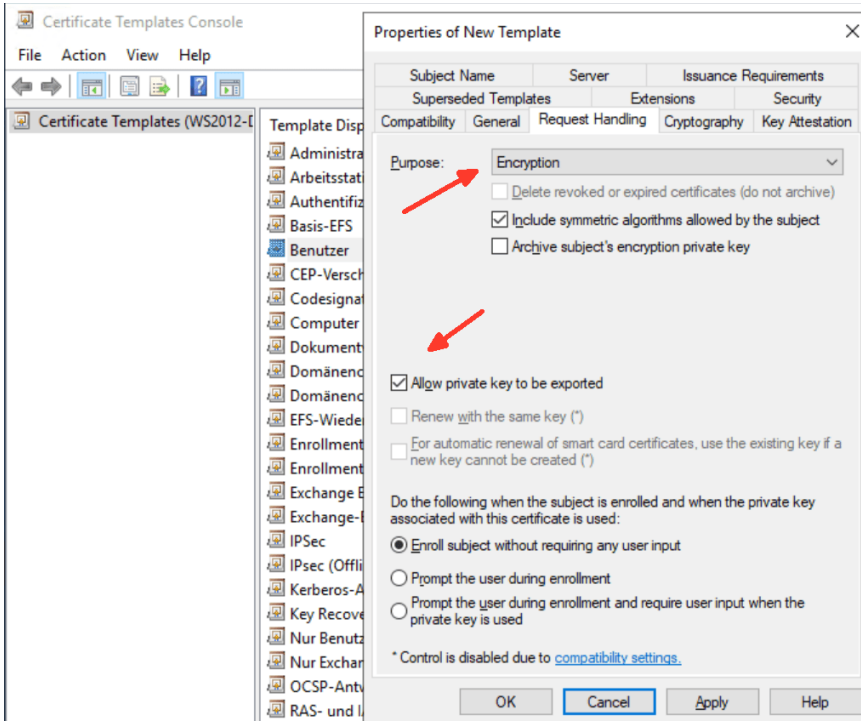
Then assign the name for the new template under the *General* tab and determine the template's period of validity.



Assign name to the new template

Next, change the purpose on the *Request Handling* tab to *Encryption*. Here, you can also allow the private key to be exported if certificates for

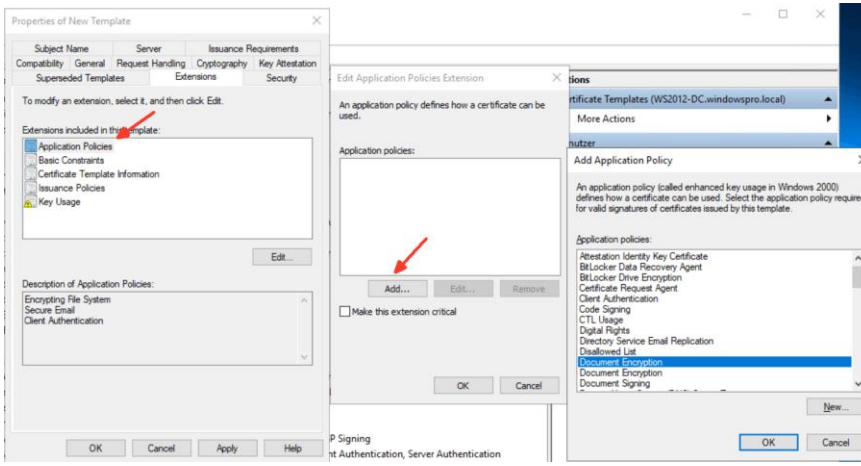
document encryption are needed on several computers to decrypt documents.



Change the purpose of the certificate template to "Encryption."

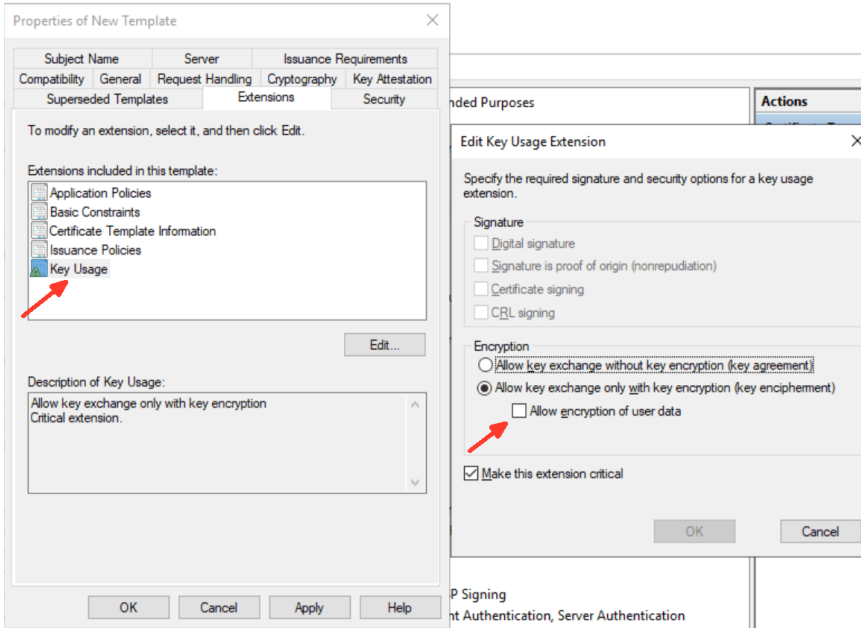
As with the .inf file shown above, the key length should be at least 2048 bits; the corresponding setting is found on the *Cryptography* tab.

Configure the necessary settings on the *Extensions* tab. Here, we edit the *Application Policies* and remove all existing entries. Instead, we add *Document Encryption*.



Document encryption is added to the application policies

By default, the new certificate is used to encrypt the CERT_KEY_ENCIPHERMENT_KEY_USAGE certificates in the .inf file, which is sufficient for the task described here. If you want to add CERT_DATA_ENCIPHERMENT_KEY_USAGE, then edit the *Key Usage* entry and select the *Allow encryption of user data* option in the next dialog box.



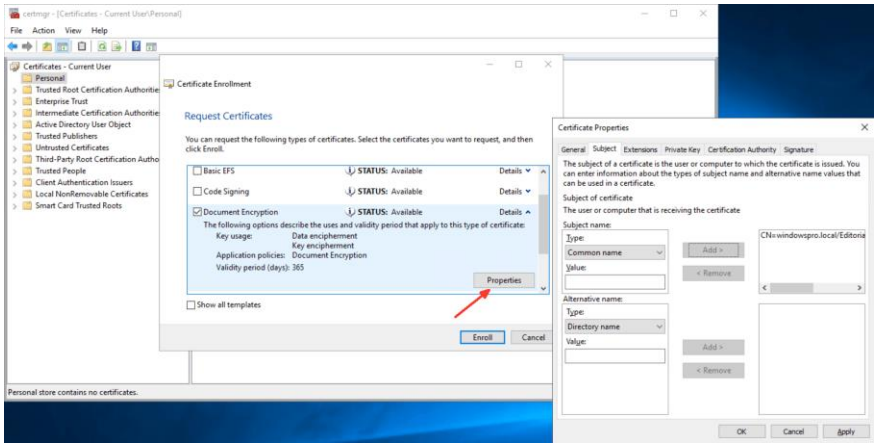
Enable encryption of user data when using keys

Finally, use the *Security* tab to make sure all users who request a certificate based on this template have the *Read* and *Register* permissions.

5.3.2 Requesting a certificate

Now you can request your certificate using *certmgr.msc*. If you can't find your new template in the list or it has a status of *Unavailable* in the extended view, then try this troubleshooting tip.

In the details, enter the subject name in the format specified in the template. Under *Private key => Key options*, make sure it is exportable, if required.



Request a certificate based on the new template

After you click *Enroll*, the new certificate should appear in the store of the *Current User*.

5.4 Encrypt event logs and files with PowerShell and GPO

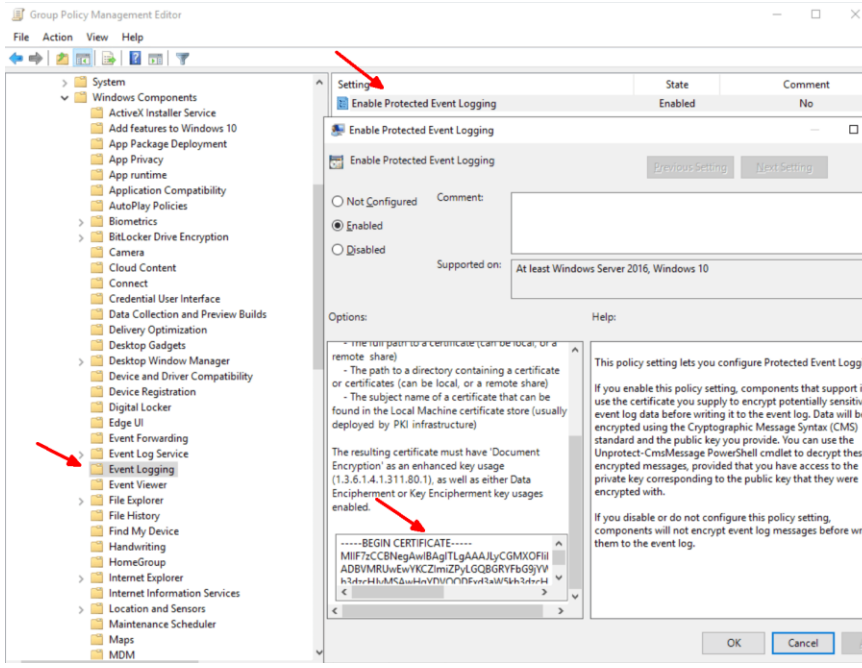
A feature introduced with Windows 10 and Server 2016 is *Protected Event Logging*, which encrypts sensitive data in the event log. It uses the open standard Cryptographic Message Syntax (CMS), which PowerShell supports with several cmdlets. You can also use them to encrypt or decrypt files.

You may wonder why you should encrypt Windows log files. This feature was triggered by the introduction of scriptblock logging in PowerShell 5, which stores all entered commands in the event log. These commands may also include credentials, which should not be visible to unauthorized persons.

5.4.1 Activation via group policies

Basically, *Protected Event Logging* is a system-wide feature that can be used by all applications and Windows services. If you activate it under Windows 10, PowerShell is currently the only user of this encryption.

To enable secure event logging, Microsoft provides a setting in Group Policy. It is called *Enable Protected Event Logging* and can be found under *Computer Configuration => Policies => Administrative Templates => Windows Components => Event Logging*.



The encryption of PowerShell entries in the event log can be enabled via group policies

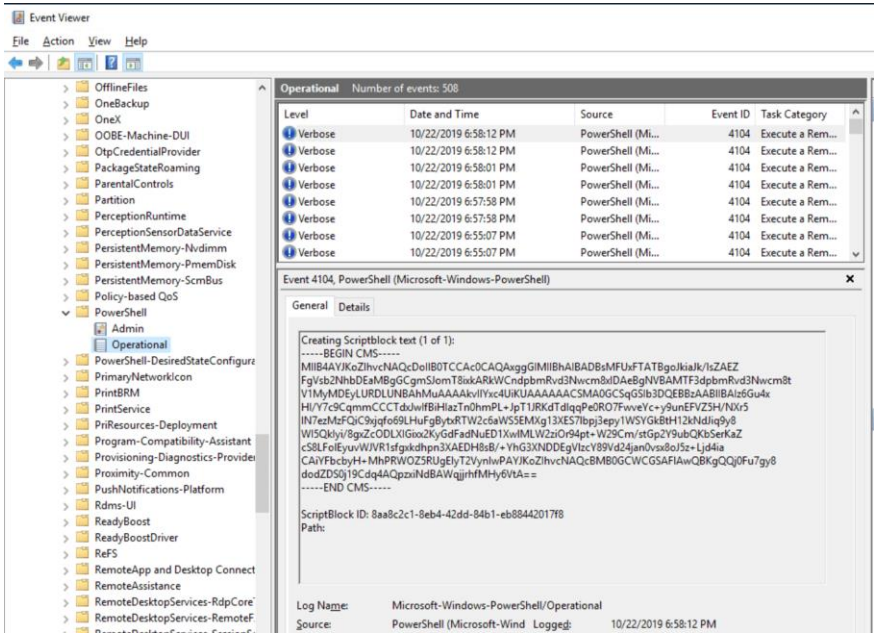
To successfully activate this setting, a certificate specifically issued for document encryption is required. Its public key is used to encode the log entries. The GPO editor accepts several ways to link the policy to the certificate.

You can store it on a file share and specify its path. If the certificate is available in the store of the local computer, the user's fingerprint will also suffice. A simple method is to export the certificate in Base64 encoded form, the contents of which can simply be copied into the text field.

5.4.2 Decrypting logs with PowerShell

Once the GPO is in effect, you can no longer read the event log history of the PowerShell commands entered on those machines. However, the

Event Viewer lacks the necessary functions to decode the logs using the private key.



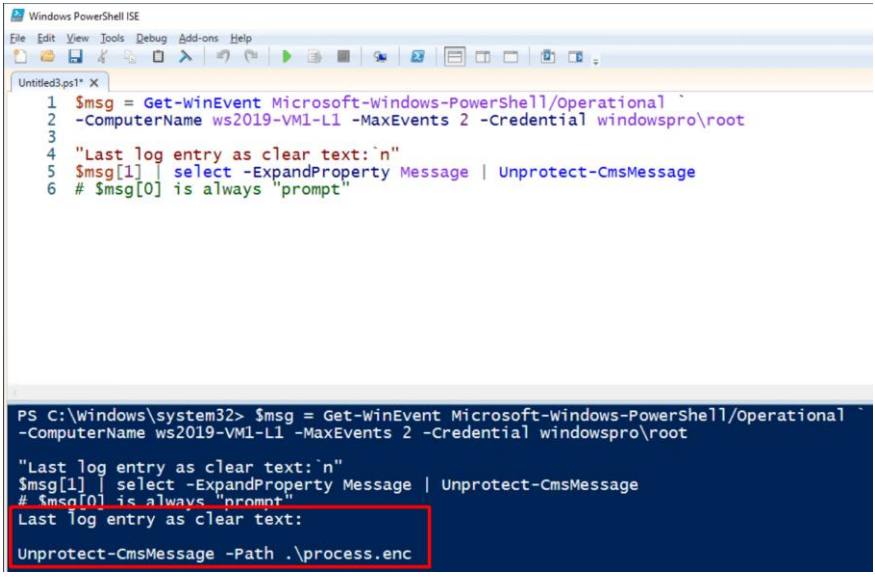
The Event Viewer only presents the encrypted entries; it cannot decode them

Therefore, you must make these log entries readable with PowerShell. The *Unprotect-CmsMessage cmdlet*, the opposite of *Protect-CmsMessage*, decrypts them.

For example, if you want to decipher the latest entry in the PowerShell log, you could retrieve it via Get-WinEvent and pipe it to *Unprotect-CmsMessage*:

```
$msg = Get-WinEvent `
Microsoft-Windows-PowerShell/Operational `
-ComputerName myPC -MaxEvents 2 -Credential domain\user

"Last log entry as clear text:"
```

```
$msg[1] | select -ExpandProperty Message |
Unprotect-CmsMessage

# $msg[0] is always "prompt"
```



Decrypting PowerShell logs with Unprotect-CmsMessage

A complete script for this purpose can be found on Emin Atac's blog (bit.ly/2DM85Gx)

The problem with script block logging is that longer command sequences are split across multiple log entries. Therefore, in this case you would have to aggregate the individual sections and then pass them to *Unprotect-CmsMessage*.

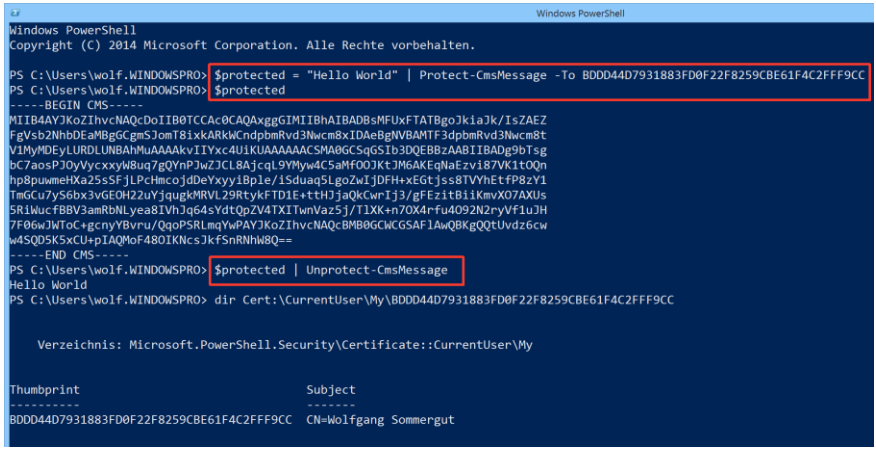
5.4.3 Encrypting files

Protect-CmsMessage can also be used to encrypt any file. If their contents are binary, then you should convert to a Base64 representation first.

Usage scenarios here may also include protecting sensitive data in scripts or password files against unauthorized access. However, this technology is certainly not intended as an alternative to an encrypting file system or even a Bitlocker.

Because PowerShell uses the cryptographic message syntax standard, you can decrypt encoded files using other tools on different platforms, such as OpenSSL on Linux (bit.ly/2E21l6Y). Therefore, this PowerShell feature is also suitable for exchanging confidential data between different operating systems.

The process is relatively simple. *Protect-CmsMessage* expects the input file via the *Path* parameter. Alternatively, you can provide the contents to be encrypted via the *Content* parameter or via a pipeline. The target file is specified via *OutFile*; otherwise, the output is stdout.



Simple application of Protect-CmsMessage and Unprotect-CmsMessage

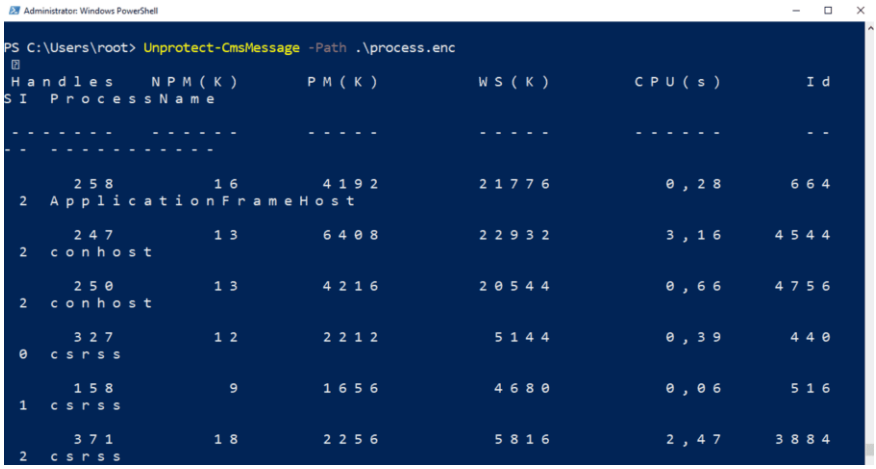
Other required information includes the certificate you want to use. The parameter *To*, which accepts the fingerprint, subject name, or path to a certificate, serves this purpose.

Conversely, *Unprotect-CmsMessage* only needs the content for decryption (via *Content* or *Path*); passing it via a pipe is also possible. The *To* parameter can be omitted if the certificate is in the local store.

5.4.4 Problems with the character set

Watch out for the character encoding of files. Otherwise, you will be surprised by a distorted result after decryption. This is the case, for example, with the following procedure:

```
Get-Process > process.txt

Protect-CmsMessage -Path process.txt -out process.enc `
-To 61F4C2FFF9CC…

Unprotect-CmsMessage -Path process.enc
```



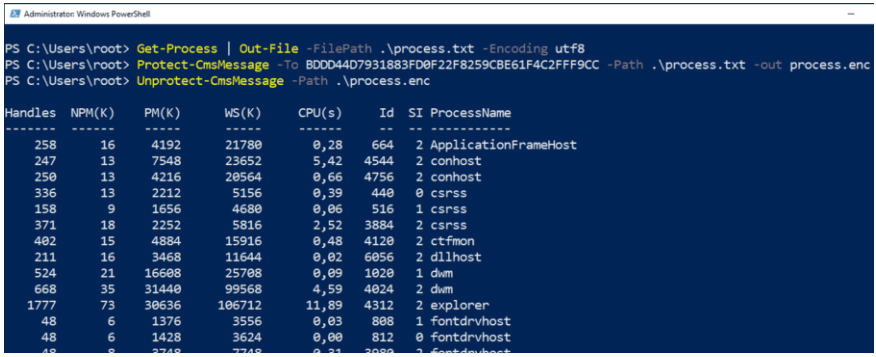
Incorrect character encoding destroys content during encryption and decryption

To avoid such unwanted effects, save the output using:

```
Get-Process | Out-File -FilePath process.txt `
-Encoding utf8
```

If you prefer the first variant with redirection to a file, then you must convert the content to the correct character set when it is read for encryption:

```
Get-Content -Raw -Encoding UTF8 process.txt |
Protect-CmsMessage -To "CN=Max White" -out .\process.enc
```



Correct decoding of encrypted data when using UTF-8

With this variant, you can take advantage of the appropriate features of *Get-Content*.

5.5 Audit PowerShell keys in the registry

The Windows registry contains numerous security-critical settings an attacker can manipulate to override important protection mechanisms. For example, an attacker can abuse it to bypass group policies. Auditing the registry helps identify such undesirable activities.

If you want to protect PowerShell against misuse and record all commands executed from the command line in a log file, a hacker probably wants to disable this function to leave no traces. To do this, he could set the value of *EnableTranscripting* to 0. This key is under:

```
HKLM\SOFTWARE\Policies\Microsoft\Windows\Pow-
erShell\Transcription
```

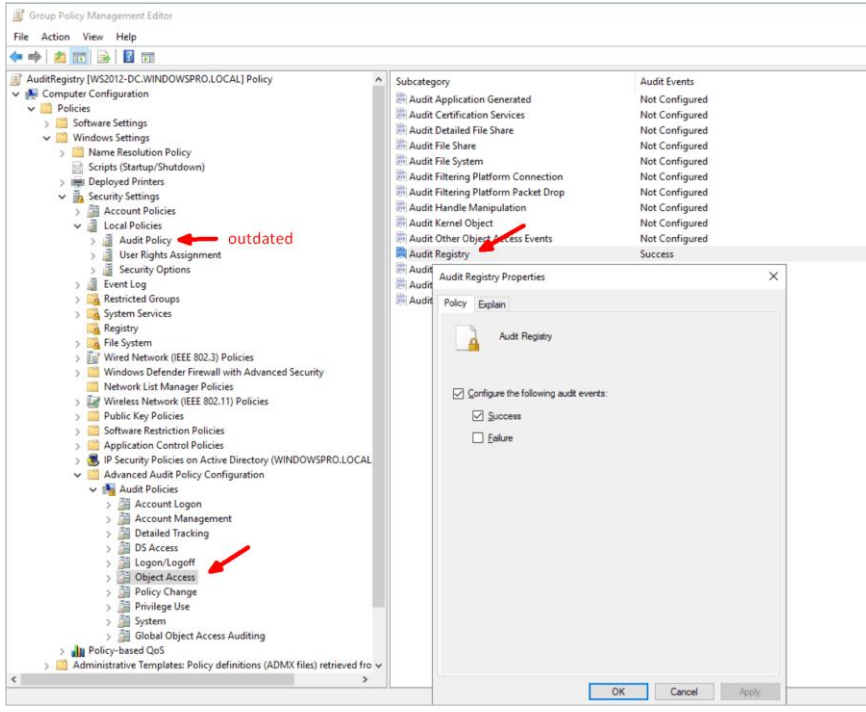
To find out about such manipulations, you should monitor the relevant keys in the registry. In our example, these would be those set by Group Policy Objects (GPOs) for PowerShell. As with auditing the file system, three measures are required:

- Enable registry monitoring via GPO
- Configure the system access control list (SACL) for the resource in question
- Analyze the event log

5.5.1 Activate registry auditing

The first step is to create a GPO and link it to the organizational unit (OU) whose machines you wish to monitor for changes to the PowerShell keys in the registry.

Next, open the new policy in the GPO editor and navigate to *Computer Configuration => Policies => Windows Settings => Security Settings => Advanced Audit Policy Configuration => Audit Policies => Object Access*. (Microsoft has deprecated the settings under *Security Settings => Local Policies => Audit Policy* since Windows 7.)



Activate auditing for registration via GPO

There you activate the *Audit Registry* setting, where you see two options: *Success* and *Failure*. Deciding whether you want to record failed, successful, or both accesses depends on the type and importance of the resource. However, you should find a balance between the relevance of the recorded events and the amount of data generated.

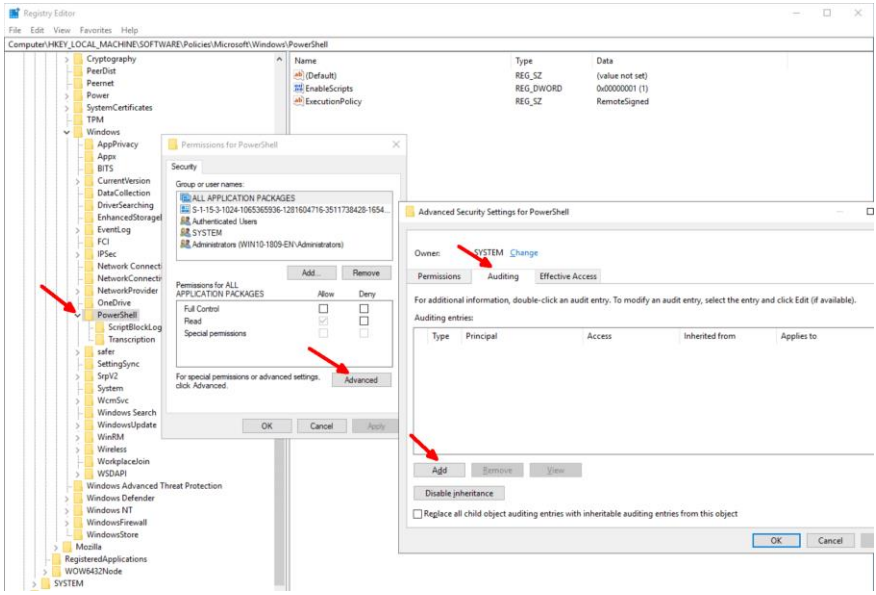
In our example, we limit ourselves only to *Success* to find out when the value of a key actually changed. Executing this command on the target computers activates the group policy:

```
gpupdate /force
```

And now you can customize the SACL for the registry key.

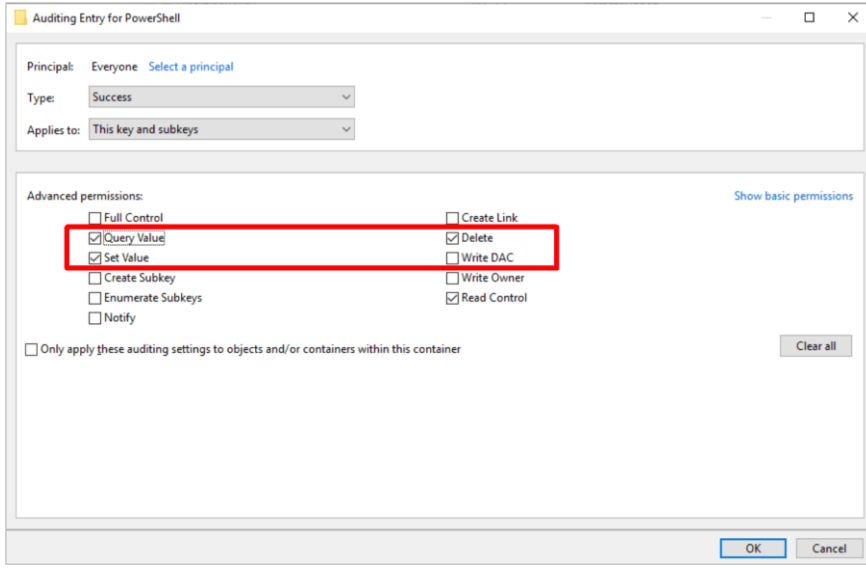
5.5.2 Setting permissions for registry keys

To do this, navigate in *regedit.exe* to the described position in the registry hive and execute the *Permissions* command from the *PowerShell* key context menu. In the subsequent dialog, click on *Advanced* and open the *Auditing* tab in the next dialog.



Editing the SACL for registry keys under PowerShell

Here you add a new entry. First, choose a security principle for tracking, such as *Everyone*. In the next step, define which activities to record. For our purpose, we select *Query Value*, *Set Value*, and *Delete* to record that a value for this key has changed.



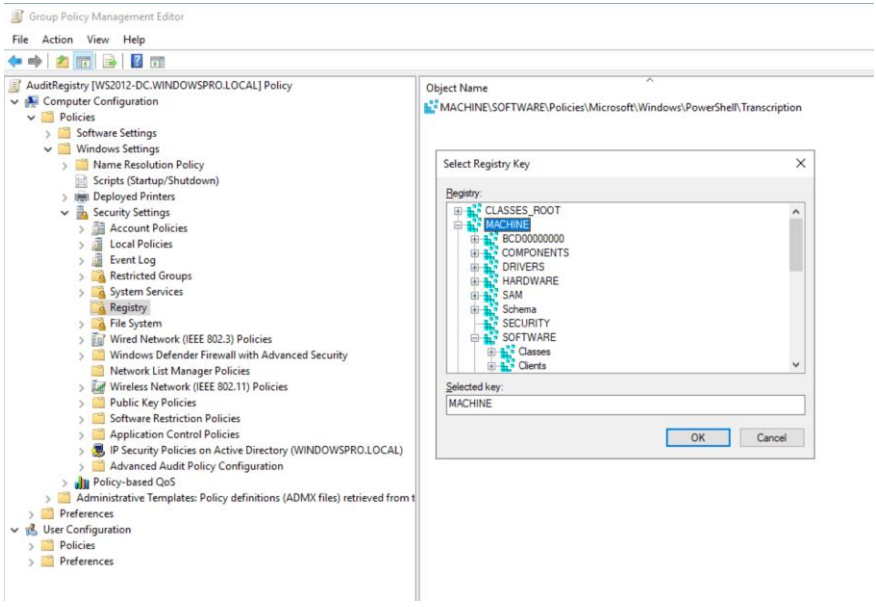
Select the type of accesses to record in the audit log

Again, you should keep in mind that monitoring *full access* may generate too much data, especially if you configure the SACL further up in the registry tree.

5.5.3 Configuring SACL via GPO

When changing the SACL of this key in the registry of many computers, it makes sense to use a GPO. You can configure the necessary setting under *Computer Configuration => Policies => Windows Settings => Security Settings => Registry*.

There you open the context menu of the container or right-click in the right panel. Then execute the *Add Key* command. In the following dialog, navigate through the registry until you reach the desired key. If this key does not exist on the local machine, you may also type the path into the input field.



You can also change the SACL of a registry key via a GPO

After selecting a key, the same security dialog opens as described above for *regedit.exe*. Therefore, the following procedure is the same as for configuring the SACL in the registry editor.

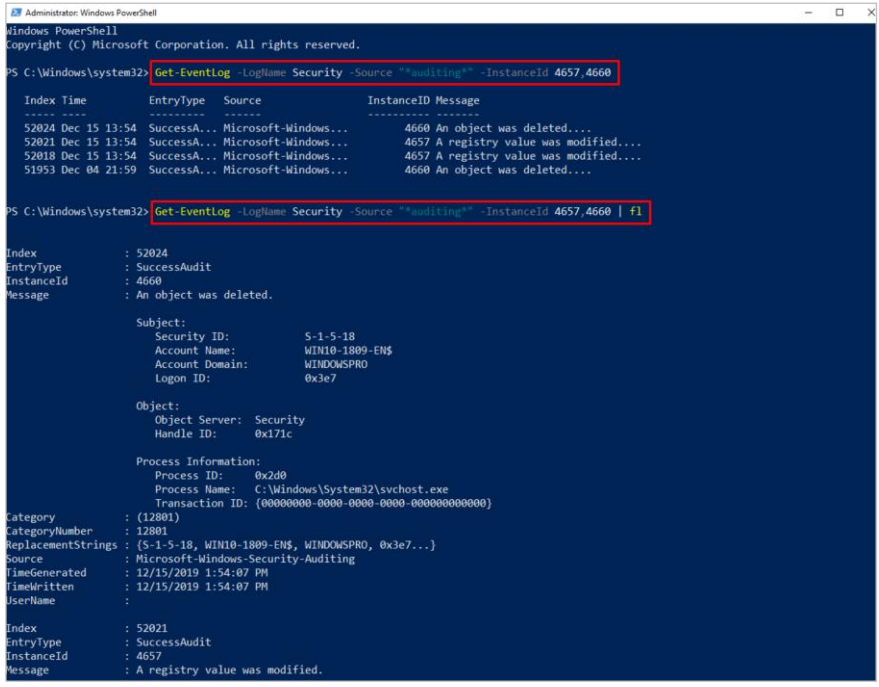
5.5.4 Evaluating the event log

Finally, you should monitor the entries in the event log to discover suspicious activities. Find these in the *Security* protocol with the IDs 4656, 4657,

4660, and 4663. As we are only interested in changes in this specific case, the *Event IDs* 4657 and 4660 are sufficient. ID 4660 represents deletion.

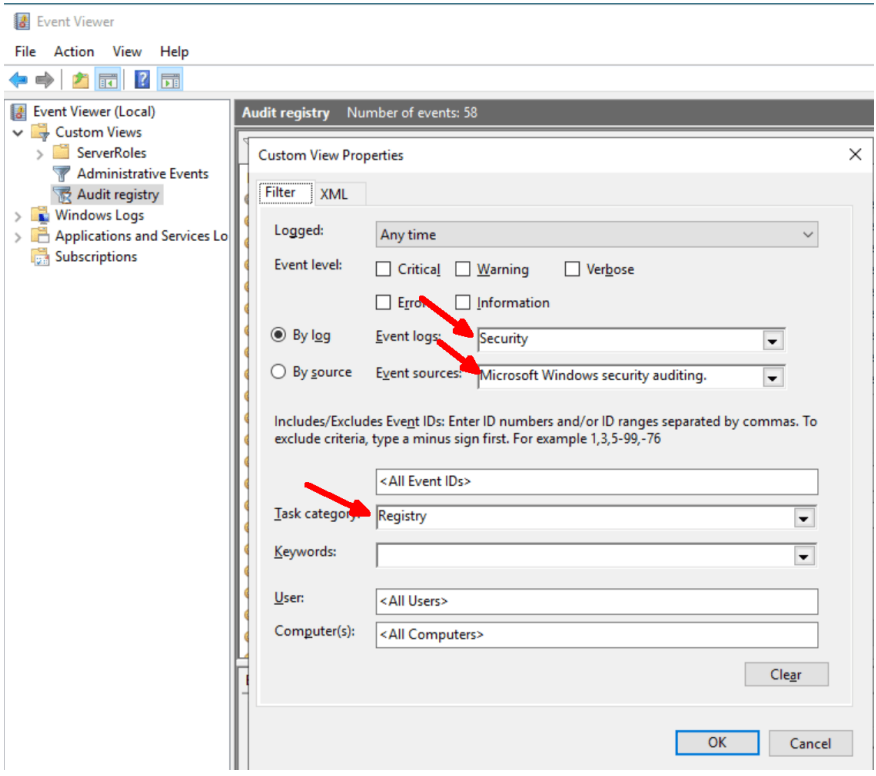
You can retrieve these logs with PowerShell as follows:

```
Get-EventLog -LogName Security -Source "*auditing*" -
InstanceId 4657,4660
```



Output audit logs for registration via PowerShell

If you prefer a GUI, you can create a user-defined view in the Event Viewer.



Set up a custom view in the Event Viewer to filter out audit logs for registration

As a filter, select *Security* under *Event logs*, *Microsoft Windows security auditing* for *By source*, and *Registry* for the *Task category*. Alternatively, you can of course also filter the view using the event IDs.

6 Improve PowerShell code

6.1 Avoiding errors using strict mode

Like other dynamic programming languages, PowerShell gives the user a lot of freedom. This simplifies the fast development of short scripts, but it also encourages sloppy programming style and all the problems resulting from it. Strict mode eliminates some typical PowerShell pitfalls.

Strict mode is not a security feature in the narrower sense, although it can be used to avoid bugs that could lead to data loss in the worst case. Its primary purpose is to prevent errors in code that is syntactically correct but leads to unwanted results. Their causes are often very difficult to track down.

6.1.1 Versions of the strict mode

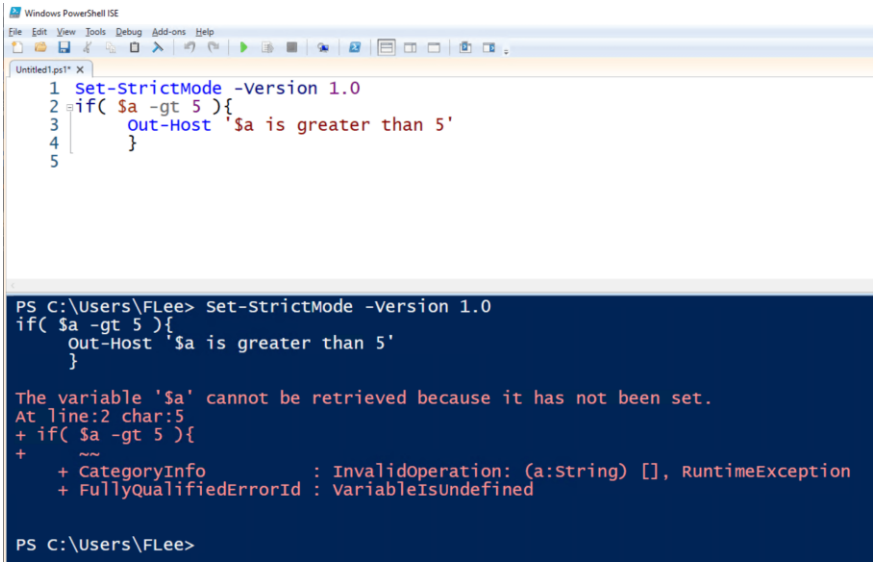
Perl has known such a mechanism for a long time, and in VBScript you can use *Option Explicit* to force variables to be declared before they are used for the first time. However, this mechanism doesn't overly limit developers and require them for example to declare data types.

While in Perl you can enable strict mode separately for variables, subs and references, PowerShell only expects a version number or the value *Off*. You pass the version number to the *Set-StrictMode* cmdlet.

6.1.2 Strict Mode 1.0

The version 1.0 prevents the use of undeclared variables:

```
Set-StrictMode -Version 1.0

if( $a -gt 5 ){

      Out-Host '$a is greater than 5'

      }
```



Strict Mode 1.0 prevents the use of undeclared variables.

Since $a is used in the if expression without a value being assigned to it, PowerShell shows an error message at this point.

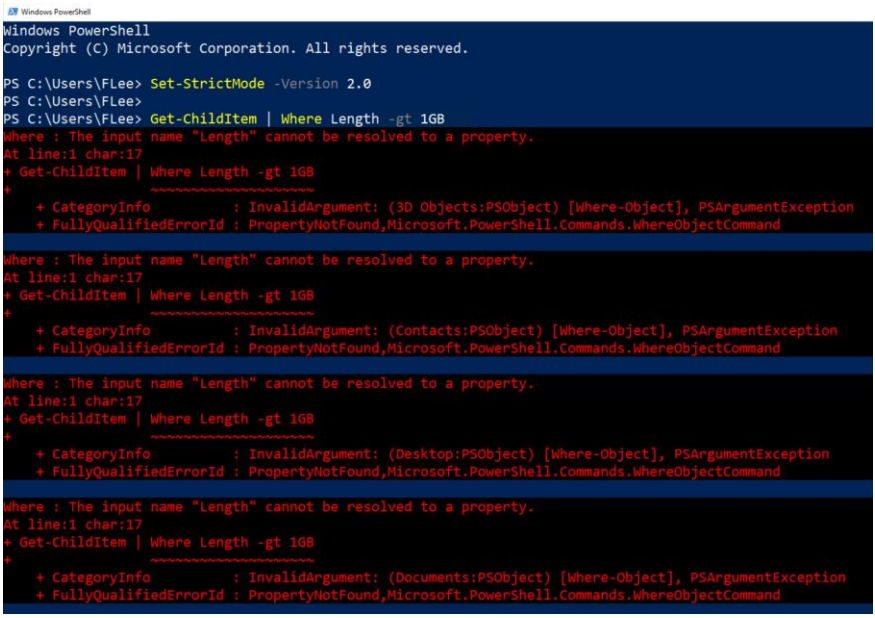
6.1.3 Strict Mode 2.0

Version 2.0 additionally checks whether non-existing properties of an object are referenced. This can happen due to a typo or because you are dealing with a mix of objects where some do not have certain properties.

An example would be if you want to display all files that exceed a certain size:

```
Get-ChildItem | Where Length -gt 1GB
```

When Strict Mode Version 2.0 is activated, this command would issue an error message for all directories because they do not have a *Length* property.



Strict Mode 2.0 prevents the use of non-existent object properties.

This is also where the ambivalent nature of this mode becomes apparent, because it triggers alarms even in harmless cases. Without Strict Mode the directories would simply not be displayed.

Rather than avoiding strict mode 2, you would have to program more defensively in this example. You could filter out the directories using the *PSIsContainer* property:

```
gci |
? {$_.PSIsContainer -eq $false -and $_.length -gt 1GB}
```

Strict mode 2.0 also helps to avoid wrong function calls. The different syntax for executing methods and functions is one of the most popular pitfalls in PowerShell, especially for those users who often deal with other programming languages.

The command

```
myfunc(1, 2, 3)
```

interprets the arguments as one array instead of three different parameters.

6.1.4 Strict Mode 3.0

Finally, there is version 3.0 of Strict Mode, but it is not documented. You will get it automatically when you invoke

```
Set-StrictMode -Version Latest
```

in PowerShell 3.0 or a higher version. But you can also specify the "3.0" explicitly here.

In addition to the criteria of the other two versions, it also checks whether elements of an array are retrieved with an invalid index. This can happen relatively easily if you iterate over the elements of an array in a loop:

```
# At least PowerShell 3.0
$array = (1,2,3)
# No error, output of $null
Set-StrictMode -Version 2.0
for ($i= 0; $i -le 3; $i++){
     $array[$i]
     }
```

```
# Error IndexOutOfRangeException
Set-StrictMode -Version 3.0
for ($i= 0; $i -le 3; $i++){
     $array[$i]
     }
```

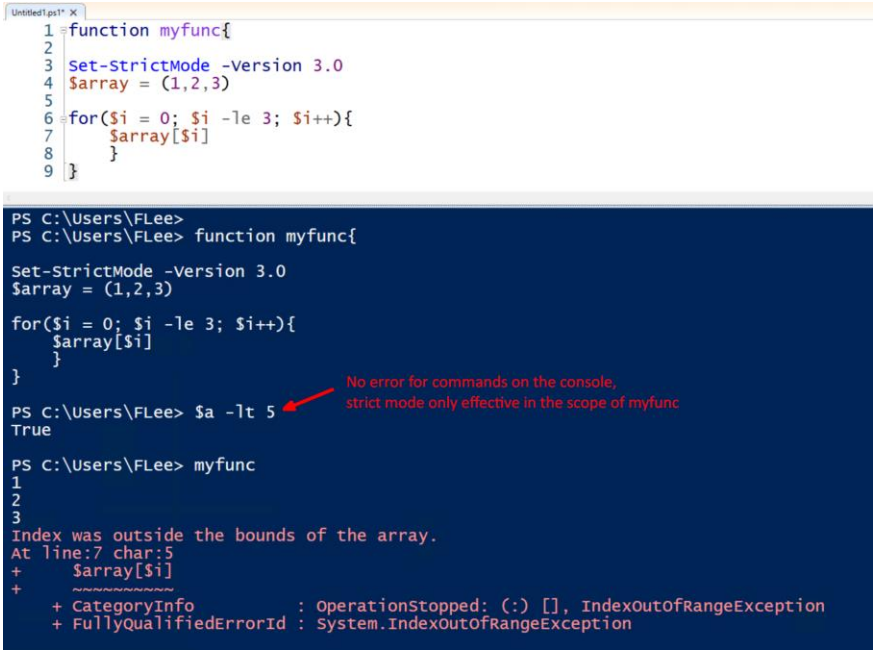
The terminating condition for the loop is

```
$i -le 3
```

and this would also reference $array[3]. With only 3 elements, the highest index is 2. Hence, Strict Mode 3.0 also acts as a bounds checker. Without it, PowerShell would output the value $null here.

6.1.5 Scope of the strict mode

Finally, it should be noted that the definition of strict mode only applies to the respective scope and all its included scopes.



The strict mode defined in the function does not apply to calls on the command line.

If you set strict mode to version 3.0 in a *function*, for example, the default setting remains on the console, i.e. switched off. Conversely, entering

```
Set-StrictMode -Version 3.0
```

on the command line will result in PowerShell checking all scripts started from there to see whether the array index is out of bounds.

6.2 Checking code with ScriptAnalyzer

The open source project *PSScriptAnalyzer* is developing a code checker that compares script code to predefined rules. They are based on the best practices for PowerShell. It can even automatically correct certain deviations.

The first versions of the code checker could be integrated into PowerShell_ISE as an add-on called Script Browser. However, this no longer works in PowerShell 5.x and the plug-in has been removed from the PS Gallery. Instead, the Analyzer is now available through the PowerShell extension of Visual Studio Code and as a stand-alone module.

6.2.1 Installation via package management

If you develop PowerShell scripts not in VSCode, but in the ISE, as most admins will probably do, then you can start the code checker from the command line. To do so you have to install the module from the PSGallery first:

```
Install-Module -Name PSScriptAnalyzer
```

As the command

```
Get-Command -Module PSScriptAnalyzer
```

shows, the module provides three cmdlets:

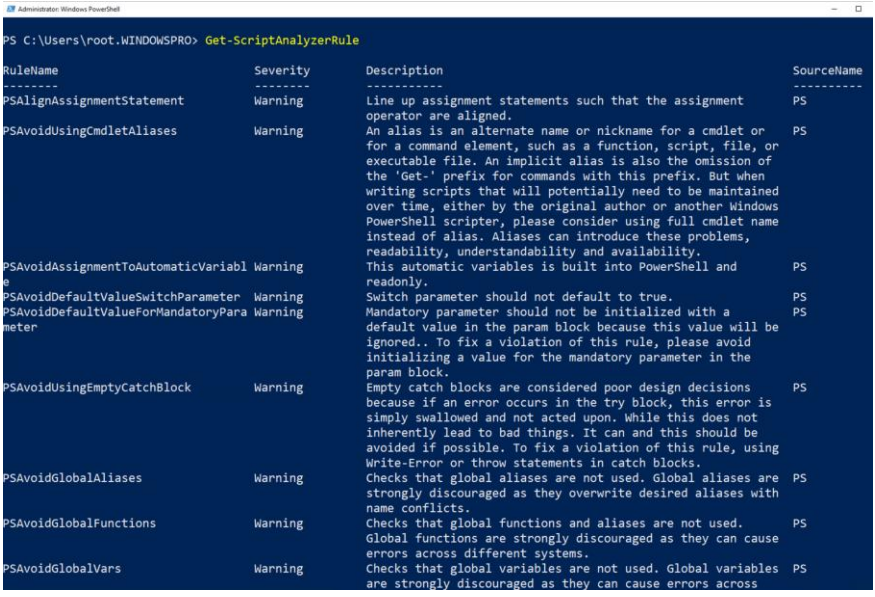
- Get-ScriptAnalyzerRule
- Invoke-ScriptAnalyzer
- Invoke-Formatter

6.2.2 Displaying the rules

The first of these cmdlets is used to display the available rules against which the code of scripts is compared. If you call it without parameters, it will show all of the currently 55 standard rules including their descriptions. A useful parameter is *Severity*, which can use the *Error and Warning* values to limit the list to serious or less serious problems:

```
Get-ScriptAnalyzerRule -Severity Error
```

This command would only show rules where a violation would be classified as a bug. You need an overview of the rules set if you want to consider only certain recommendations or exclude others during the review.



View the default rules for ScriptAnalyzer using Get-ScriptAnalyzerRule.

Several rules have been added to the latest versions:

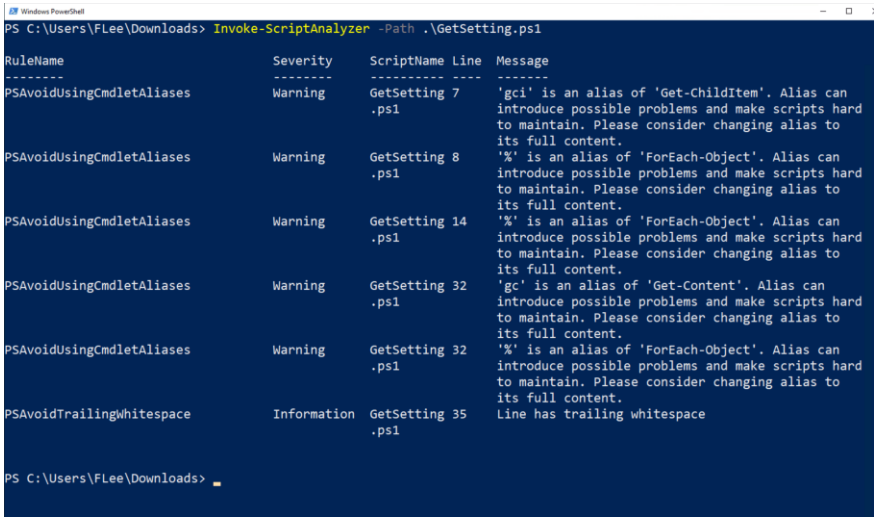
- AvoidAssignmentToAutomaticVariable: This is to prevent developers from assigning values to automatic variables such as $ _.
- PossibleIncorrectUsageOfRedirectionOperator: This is mainly intended for developers who often use other programming languages, where a > or < serve as a comparison operators for greater or smaller. PowerShell uses -gt or -lt instead. The characters > and < are reserved for redirection.
- PossibleIncorrectUsageOfAssignmentOperator: Checks for the possibly wrong usage of the assignment operator. This could happen, for example, if Basic developers validate an expression for equality.
- AvoidTrailingWhiteSpace: The rule warns of spaces at the end of a line of code. They could become a problem if line breaks occur within a statement.

6.2.3 Checking script code

The actual checking of code is done with the help of *Invoke-ScriptAnalyzer*. In most cases, you pass the cmdlet the name of a script file that you want to check:

```
Invoke-ScriptAnalyzer -Path .\MyPSScript.ps1
```

The parameters *IncludeRules* and *ExcludeRules* can be used to explicitly include or exclude certain rules. If you specify several rules here, then they should be separated by a comma. Simple warnings could be suppressed, for example, by assigning the value *Error* to the *Severity* parameter.



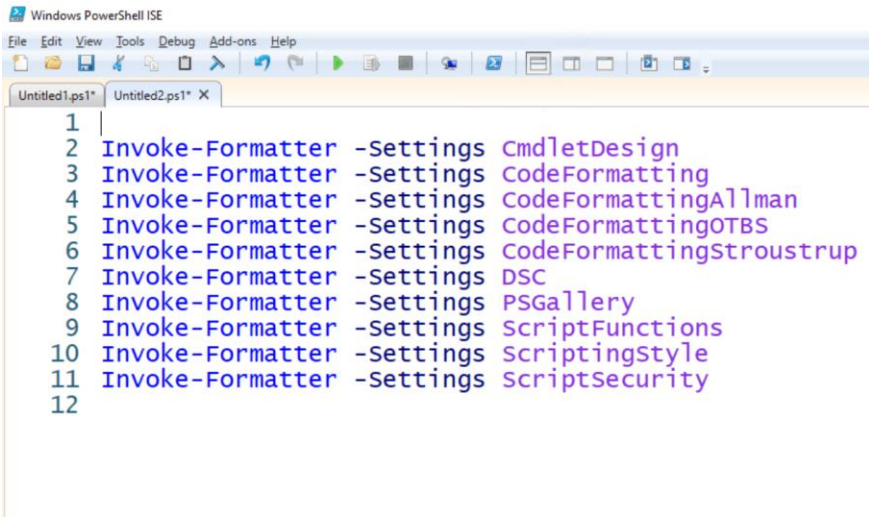
In this example, ScriptAnalyzer warns about using an alias in a script.

A new option in version 1.17.1 is *Fix*, which can automatically correct certain deviations from the commonly used rules. This applies, for example, to the use of aliases for cmdlets. In some cases, script authors have to edit such corrections manually, for example when converting plain text to a secure string.

If you only want to check a code fragment and not an entire script file, use the *ScriptDefinition* parameter instead of *Path* and pass the code to it as a value. In this case, the *Fix* switch is not available for obvious reasons.

6.2.4 Formatting scripts

Finally, *Invoke-Formatter* is the third cmdlet that comes with the module. As the name suggests, script authors can use it to tidy up the formatting of the code. There are several conventions to choose from, which can be selected via the *Settings* parameter using auto-completion.



Formatting options of Invoke-Formatter

It only accepts PowerShell code via the *ScriptDefinition* parameter, so you may have to read the content of a script file via *Get-Content-Raw* before you pass it to the formatter.

A detailed documentation of the module can be found on Github (bit.ly/2rSLdil).

7 More security with ScriptRunner

7.1 PowerShell management solution

In many organizations only a few selected experts use the capabilities of PowerShell. The reasons for this are:

- PowerShell know-how is not widely available
- No central management of PowerShell scripts
- No secure credential management
- Delegation fails due to security and authorization reasons

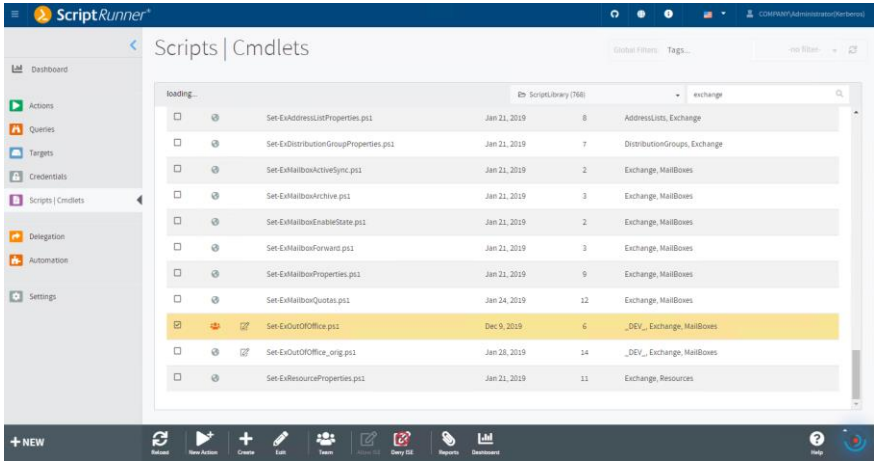
ScriptRunner transforms PowerShell into a solution that benefits your entire organization by making it much easier to develop, manage and delegate scripts. As such, it takes care of the necessary security aspects. Thereby PowerShell can also be used as a tool for the administration of heterogeneous systems.

7.2 Five steps to safe automation and delegation

7.2.1 Central storage of all PowerShell scripts

When centralizing the scripts, it is important to store them in a well-structured way so that they can be easily retrieved later.

They can be separated by target systems such as Exchange, Office 365 or Azure, or by target groups such as helpdesk or end users. Incidentally, the folder names automatically represent tags, which allow you to easily filter scripts.



ScriptRunner provides a central repository for all PowerShell scripts in the company.

Besides scripts, PowerShell modules are also subject to central administration. Therefore, you must, for example install modules for Office 365, Azure or VMware only once and after that, they will be available for all further activities.

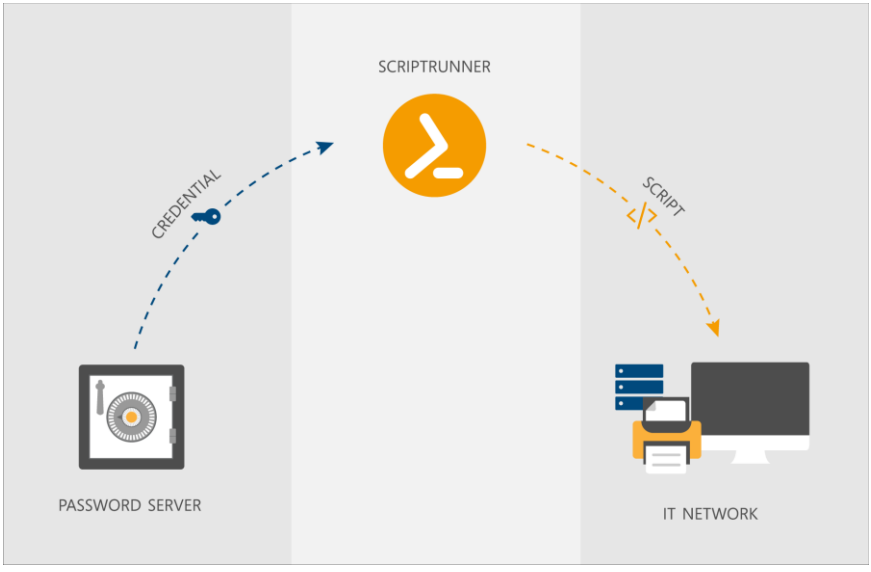
Centralization is also an important step towards standardization, because it ensures that the most current version of a specific script is always used

for all tasks. An automatic synchronization of the scripts with code management systems such as GitHub or TFS is also possible.

7.2.2 Secure credential management

Securely managing usernames and passwords for script execution is one of the biggest challenges. ScriptRunner allows you to store this information centrally and securely. For this purpose, the Windows credential store is available on the ScriptRunner server, and password servers from CyberArk, Pleasant and Thycotic are also supported.

This allows you to manage all credentials in a central repository, which makes administration much easier, especially when using multiple ScriptRunner instances.



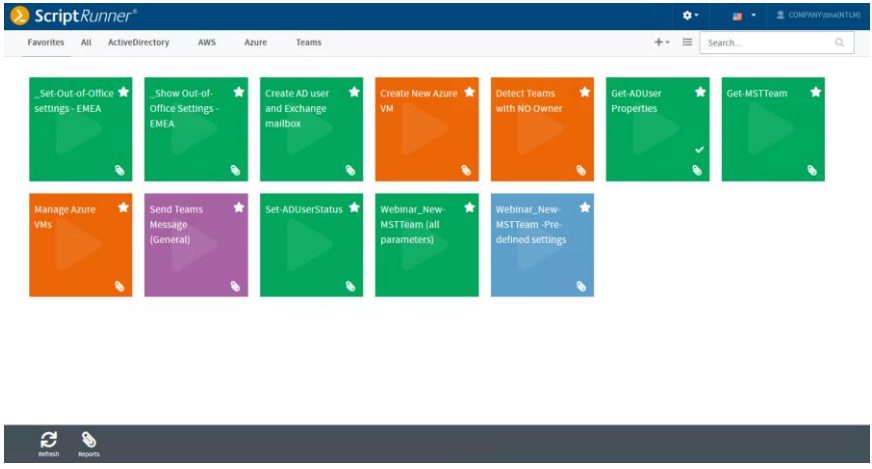
For credentials management, ScriptRunner integrates password servers from multiple vendors

7.2.3 Convenient web-based user interface

All components such as scripts, credentials and schedules are managed via the AdminApp. The DelegateApp or Self-ServiceApp are available for executing the scripts manually. Plausibility checks, dynamic selection lists, and preconfigured default values reliably prevent users from making incorrect entries.

This makes it very easy and safe for helpdesk staff and end users to perform recurring tasks with the help of scripts. PowerShell knowledge and experience with the console are not required.

The input masks are automatically generated dynamically from the synopsis and parameter block of the respective PowerShell scripts. The time-consuming programming and maintenance of user interfaces is therefore not necessary.



PowerShell scripts can be deployed to users through a self-service portal.

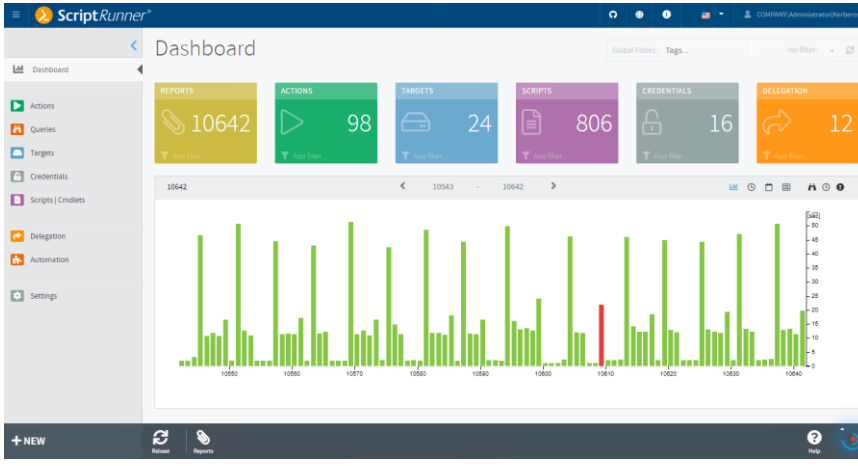
7.2.4 Executing and monitoring all PowerShell scripts centrally

By processing scripts centrally, ScriptRunner provides a complete audit trail of all PowerShell activities. Three execution options are available:

- Manually by a user using the ScriptRunner DelegateApp
- Scheduled for regular tasks
- Event-controlled by third-party systems

With the help of the ScriptRunner dashboard, you will always have an overview of your entire PowerShell landscape. Information about processing times, possible errors or unreachable backend systems can be easily retrieved.

In addition, detailed log files, Windows event log entries and Windows performance counters are available for analysis.

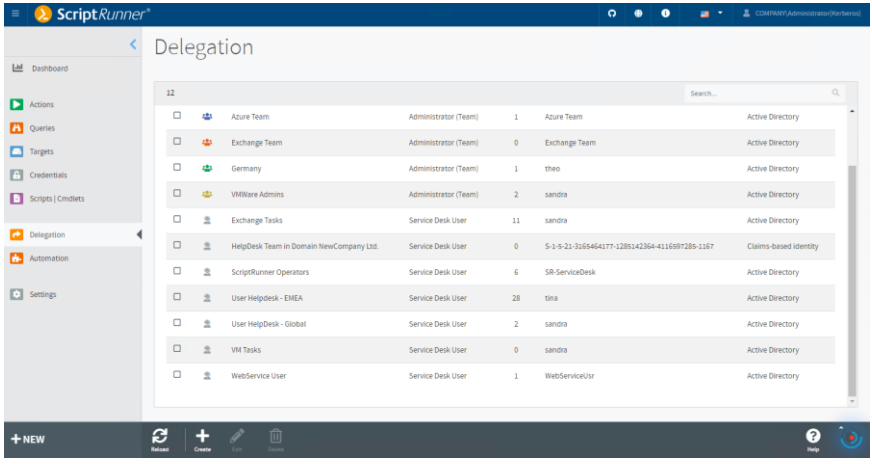


The ScriptRunner dashboard provides an overview of all scripts and its associated resources.

7.2.5 Secure delegation to helpdesk and end users

With the points mentioned above, the prerequisites are now basically in place to implement recurring tasks with PowerShell safely and easily. Delegating these tasks to employees in areas outside of IT administration, however, poses an additional challenge. How do we ensure that these employees do not need administrative permissions in the respective backend systems, such as Active Directory, Exchange, VMware or Office 365, for execution?

ScriptRunner executes scripts with the help of the central service accounts or service principals. The users are granted access to the desired actions via the delegation in ScriptRunner and therefore do not need any elevated privileges. Helpdesk staff or end users remain standard users of the domain and can still perform delegated tasks without security concerns.

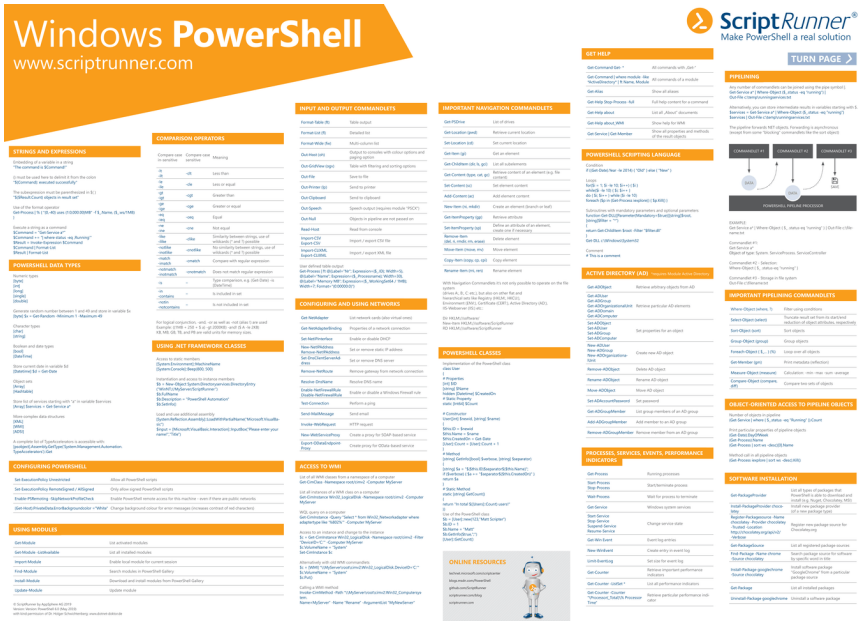


With ScriptRunner, administrative tasks can be securely delegated to standard users.

7.3 Additional information

Would you like to learn more about ScriptRunner? Then visit our homepage at www.scriptrunner.com

On our PowerShell poster, you will find all important commands and cmdlets at a glance. Get your free copy at www.scriptrunner.com/poster



The PowerShell poster contains all of the commands and important cmdlets.

You can read more about PowerShell related security topics on our blog:

https://www.scriptrunner.com/en/blog/security-powershell-scripting/

Password servers can also provide additional security. More about their use in the TechSnips video: http://y2u.be/6gfubJAs-RA