11

AppLocker, Application Control, and Code Signing

In an enterprise environment, it is critical to keep control over what software is installed and what software is being kept out of the environment – not only to keep an overview of what software is available but also to help fight against threats such as malicious scripts or malware such as ransomware.

But how can code signing and application control help you secure your environment in a better way and how can it be implemented? What do you need to do when planning for implementing an application control solution and what built-in application control solutions are available on Windows operating systems?

We'll explore this and much more in this chapter about AppLocker, application control, and code signing. In this chapter, you will get a deeper understanding of the following topics:

- Preventing unauthorized script execution with code signing
- Controlling applications and scripts
- Getting familiar with Microsoft AppLocker
- Exploring Windows Defender Application Control

Technical requirements

To get the most out of this chapter, ensure that you have the following:

- PowerShell 7.3 and above
- Installed Visual Studio Code
- A virtual machine running Windows 10 or above for test purposes
- Access to the GitHub repository for **Chapter11**: https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/tree/master/Chapter11

Preventing unauthorized script execution with code signing

If you want to verify that the executed script is legit code and is allowed to be executed by your company, you want to implement a proper code-signing strategy. It's a brilliant way to protect your regularly executed scripts against tampering – or at least if someone were to tamper with your scripts, they would not be executed if your environment is configured in the right way.

It's important to note that dynamic runtimes can pose a common blind spot when implementing application control policies. While PowerShell made a significant impact to ensure that the PowerShell runtime can be restricted by application control rules, other dynamic runtimes such as Python, Node, Perl, PHP, and more may still allow you to run unrestricted code, which might present a vulnerability if it's not managed appropriately. If other dynamic runtimes are not needed on your clients, it's better to block them or restrict them as much as possible to maintain a strong security posture.

The WSH language family has implemented application control awareness in a quite straightforward manner: they simply prevent the execution of any scripts that are not permitted by the policy.

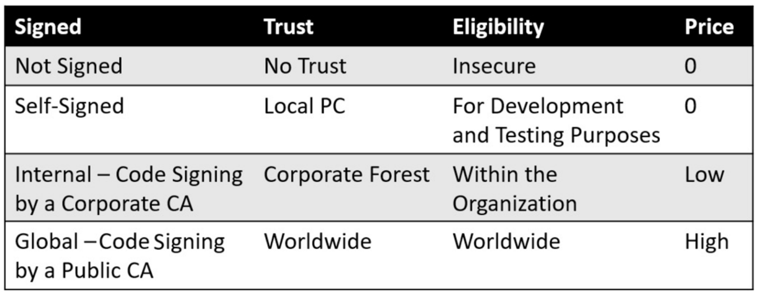
When we talked about **execution policies** in earlier chapters, such as *Chapter 1*, *Getting Started with PowerShell*, we looked at the **AllSigned** or **RemoteSigned** parameters. If **AllSigned** is configured, all unsigned PowerShell scripts are prevented from running – if **RemoteSigned** is configured, only local unsigned scripts are allowed. Of course, the execution policy can be bypassed at any time as it's not a security boundary – however, this prevents your users from unintentionally running scripts they don't know.

Combining code signing with other tools such as AppLocker or **WDAC** is powerful as you can ensure that no other scripts except for the configured signed ones are allowed in your infrastructure.

But to start with code signing, we first need a certificate to sign the code with. There are several options as to what kind of certificate you can use. You could either use a self-signed certificate or a corporate one (either on a forest or a public level) that your company paid for.

Self-signed certificates are usually for testing purposes only and if you want to take your code-signing infrastructure into production, you should at least consider using a certificate signed by your corporate **certificate authority** (**CA**) to make your deployment more secure.

The following figure should provide you with an overview of some different scenarios when it comes to code signing:



Signed	Trust	Eligibility	Price
Not Signed	No Trust	Insecure	0
Self-Signed	Local PC	For Development and Testing Purposes	0
Internal – Code Signing by a Corporate CA	Corporate Forest	Within the Organization	Low
Global – Code Signing by a Public CA	Worldwide	Worldwide	High

Figure 11.1 – Overview of the different possibilities of code-signing certificates

In this chapter, we will use a self-signed certificate to sign our scripts – please make sure you adjust your certificate if you want to use it in production.

A self-signed certificate is only valid on your local computer and can be created using the **New-SelfSignedCertificate** cmdlet. In earlier days, **makecert.exe** was used to create self-signed certificates, but ever since **New-SelfSignedCertificate** was introduced with Windows 8, you can simply create self-signed certificates and sign scripts using PowerShell.

Certificates created using this cmdlet can be stored either in the current user's personal certificate store by going to **Certificates** | **Current User** | **Personal** (**Cert:\CurrentUser\My**) or the local machine's personal certificate store by going to **Certificates** | **Local Computer** | **Personal** (**Cert:\LocalMachine\My**). Certificates that are created in the local computer's certificate store are available computer-wide, while the ones created in the current user's store are scoped to the current user only.

Let's create a self-signed certificate and add it to the computer's root certificate store, as well as to the computer's **Trusted Publishers** store. First, we must create a new certificate called **"Test Certificate"** in the local machine's certificate store and save the output in the **$testCert** variable. We will need this variable later to register the **authenticode certificate**:

```
> $testCert = New-SelfSignedCertificate -Subject "Test Certificate" -CertStoreLocation Cert:\LocalMachin
```

Once we've done this, we will add the authenticode certificate to our computer's root certificate store. A root certificate store is a list of trusted root CA certificates, so every certificate in this store will be trusted.

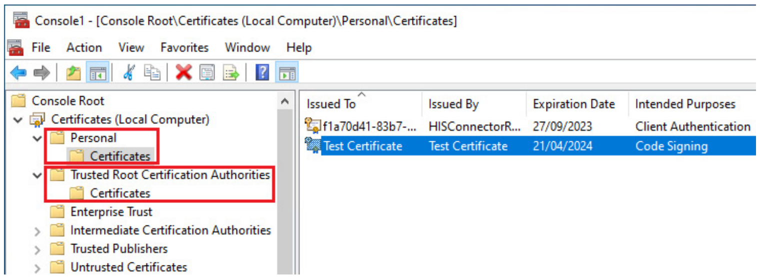
We must move the newly created certificate from the intermediate CA store to the **root certificate store**:

```
> Move-Item Cert:\LocalMachine\CA\$($testCert.Thumbprint) Cert:\LocalMachine\Root
```

Now, your certificate should be available in two different locations:

- **The local machine's personal certificate store**: This certificate will be used as the code-signing certificate.
- **The local machine's root certificate store**: Adding the certificate to the machine's root certificate store ensures that the local computer trusts certificates in the personal as well as the **Trusted Publishers** certificate store.

You can verify that all the certificates are in the right place by either using PowerShell or by using **mmc** with the local computer's certificate snap-in (run **mmc**, add the **Certificates** snap-in, and add the local computer scope), as shown in the following screenshot:


Console1 - [Console Root\Certificates (Local Computer)\Personal\Certificates]
File Action View Favorites Window Help
Console Root
Certificates (Local Computer)
Personal
Certificates
Trusted Root Certification Authorities
Certificates
Enterprise Trust
Intermediate Certification Authorities
Trusted Publishers
Untrusted Certificates

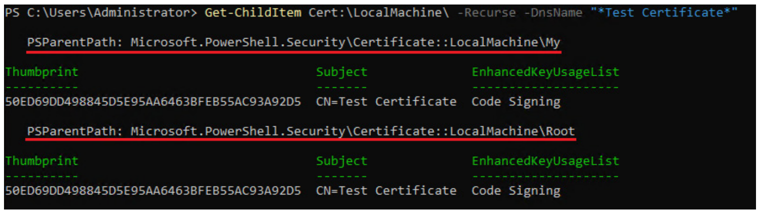
Issued To	Issued By	Expiration Date	Intended Purposes
f1a70d41-83b7-...	HISConnectorR...	27/09/2023	Client Authentication
Test Certificate	Test Certificate	21/04/2024	Code Signing

Figure 11.2 – Looking for the newly created Test Certificate

If you want to use PowerShell to check if all the certificates were created, run the following command:

```
> Get-ChildItem Cert:\LocalMachine\ -Recurse -DnsName "*Test Certificate*"
```

You can see the output of this command in the following screenshot:



```
PS C:\Users\Administrator> Get-ChildItem Cert:\LocalMachine\ -Recurse -DnsName "*Test Certificate*"

   PSParentPath: Microsoft.PowerShell.Security\Certificate::LocalMachine\My

Thumbprint                                Subject               EnhancedKeyUsageList
----------                                -------               --------------------
50ED69DD498845D5E95AA6463BFEB55AC93A92D5  CN=Test Certificate   Code Signing

   PSParentPath: Microsoft.PowerShell.Security\Certificate::LocalMachine\Root

Thumbprint                                Subject               EnhancedKeyUsageList
----------                                -------               --------------------
50ED69DD498845D5E95AA6463BFEB55AC93A92D5  CN=Test Certificate   Code Signing
```

Figure 11.3 – Verifying that all the certificates are in the right place

Now that we have created our local certificate, we can start self-signing scripts using the **Set-AuthenticodeSignature** cmdlet.

For this example, I am reusing the **HelloWorld.ps1** PowerShell script that we created in ***Chapter 1***, *Getting Started with PowerShell*, which can be downloaded from this book's GitHub repository: **https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter01/HelloWorld.ps1**.

Save the script under **C:\tmp\HelloWorld.ps1**.

If you still have the **$testCert** variable available in your session, which we used earlier when creating the certificate, you can, of course, reuse it, but most of the time, when you want to sign a script, time has already passed and you've closed the session so that the variable isn't available for you to use.

Therefore, first, assign the certificate to a variable that you will use to sign your script:

```
> $signingCertificate = Get-ChildItem Cert:\LocalMachine\ -Recurse -DnsName "*Test Certificate*"
```

Make sure you specify the correct name of the certificate that you created earlier.

To ensure that the signature on the file remains valid, even after the certificate expires after a year, it is important to use a trustworthy timestamp server when signing the script. You can do this using **Set-**

AuthenticodeSignature. The timestamp server adds a timestamp to the signed code that indicates the exact date and time when the code was signed. This timestamp is used to prove that the code was signed before the certificate expired, even if the certificate has since expired.

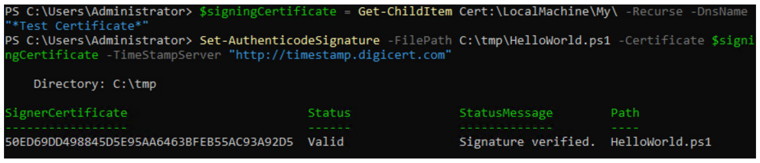
Therefore, it is recommended to always use a reliable and well-known timestamp server to ensure the longevity and authenticity of your signed code. The **Time-Stamp Protocol** (**TSP**) standard is defined in **RFC3161** and you can read more about it here: https://www.ietf.org/rfc/rfc3161.txt.

There's a great (but of course non-complete) list that's been published by David Manouchehri that you can use to choose your preferred timestamp server: https://gist.github.com/Manouchehri/fd754e402d98430243455713efada710.

For our example, I am using the **http://timestamp.digicert.com** server:

```
> Set-AuthenticodeSignature -FilePath "C:\tmp\HelloWorld.ps1" -Certificate $signingCertificate -TimeStam
```

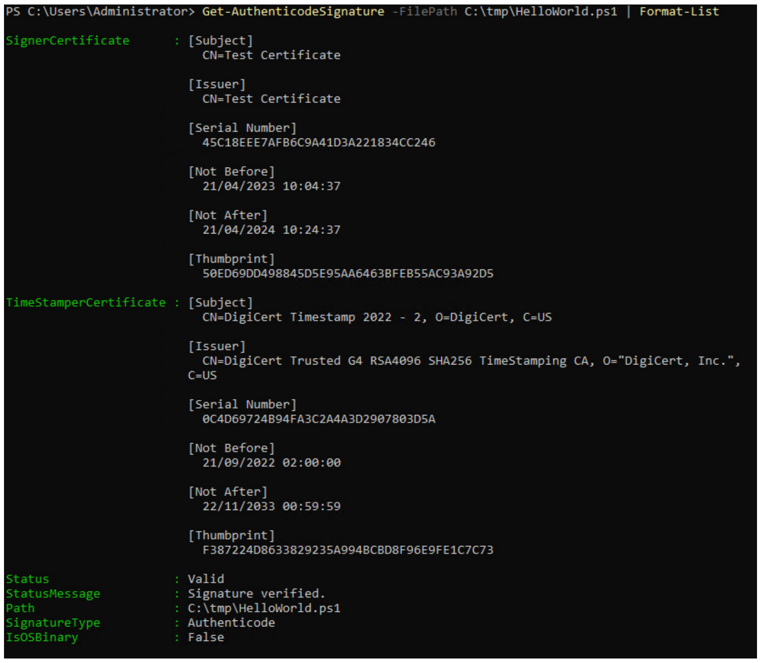
Once the script has been signed successfully, the output will look similar to the following:



```
PS C:\Users\Administrator> $signingCertificate = Get-ChildItem Cert:\LocalMachine\My\ -Recurse -DnsName
"*Test Certificate*"
PS C:\Users\Administrator> Set-AuthenticodeSignature -FilePath C:\tmp\HelloWorld.ps1 -Certificate $signi
ngCertificate -TimeStampServer "http://timestamp.digicert.com"

    Directory: C:\tmp

SignerCertificate                         Status              StatusMessage       Path
-----------------                         ------              -------------       ----
50ED69DD498845D5E95AA6463BFEB55AC93A92D5  Valid               Signature verified.  HelloWorld.ps1
```

Figure 11.4 – Script signed successfully

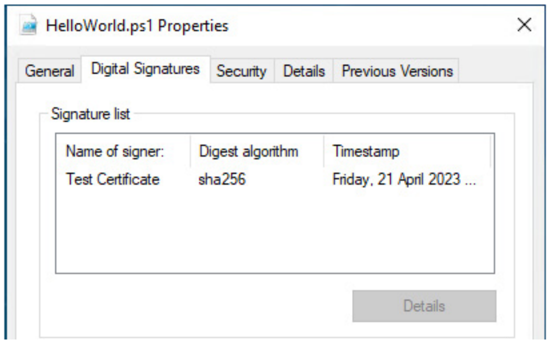
You can verify that a script has been signed by using the **Get-AuthenticodeSignature -FilePath C:\tmp\HelloWorld.ps1** | **Format-List** command, as shown in the following screenshot:



```
PS C:\Users\Administrator> Get-AuthenticodeSignature -FilePath C:\tmp\HelloWorld.ps1 | Format-List

SignerCertificate      : [Subject]
                           CN=Test Certificate

                         [Issuer]
                           CN=Test Certificate

                         [Serial Number]
                           45C18EEE7AFB6C9A41D3A221834CC246

                         [Not Before]
                           21/04/2023 10:04:37

                         [Not After]
                           21/04/2024 10:24:37

                         [Thumbprint]
                           50ED69DD498845D5E95AA6463BFEB55AC93A92D5

TimeStamperCertificate : [Subject]
                           CN=DigiCert Timestamp 2022 - 2, O=DigiCert, C=US

                         [Issuer]
                           CN=DigiCert Trusted G4 RSA4096 SHA256 TimeStamping CA, O="DigiCert, Inc.",
                         C=US

                         [Serial Number]
                           0C4D69724B94FA3C2A4A3D2907803D5A

                         [Not Before]
                           21/09/2022 02:00:00

                         [Not After]
                           22/11/2033 00:59:59

                         [Thumbprint]
                           F387224D8633829235A994BCBD8F96E9FE1C7C73

Status                 : Valid
StatusMessage          : Signature verified.
Path                   : C:\tmp\HelloWorld.ps1
SignatureType          : Authenticode
IsOSBinary             : False
```

Figure 11.5 – Verifying that a file has been signed

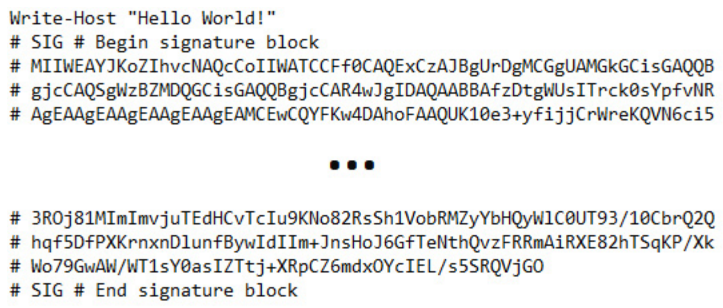
But this is not the only way to verify that a file has been signed. If you right-click on a signed file and open its properties, under the **Digital Signatures** tab, you will see that the certificate you used for signing was added:



Name of signer:	Digest algorithm	Timestamp
Test Certificate	sha256	Friday, 21 April 2023 ...

Figure 11.6 – Verifying that a file has been signed using file properties

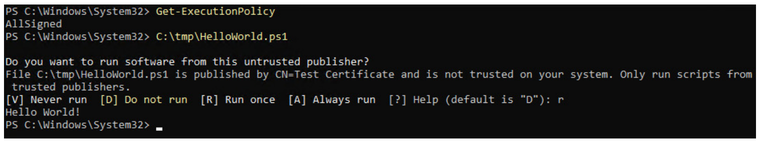
Also, if you open the newly signed script, you will see that its content has changed: instead of only the code, you will see the signature as well – introduced by # **SIG** # **Begin signature block** and closed out by # **SIG** # **End signature block** and in between a huge signature block. As shown in the following screenshot, I have shortened the signature block as the signature would be too big to show as a figure in this book:



```
Write-Host "Hello World!"
# SIG # Begin signature block
# MIIWEAYJKoZIhvcNAQcCoIIWATCCFf0CAQExCzAJBgUrDgMCGgUAMGkGCisGAQQB
# gjcCAQSgWzBZMDQGCisGAQQBgjcCAR4wJgIDAQAABBAfzDtgWUsITrck0sYpfvNR
# AgEAAgEAAgEAAgEAAgEAMCEwCQYFKw4DAhoFAAQUK10e3+yfijjCrWreKQVN6ci5

                              •••

# 3ROj81MImImvjuTEdHCvTcIu9KNo82RsSh1VobRMZyYbHQyW1C0UT93/10CbrQ2Q
# hqf5DfPXKrnxnDlunfBywIdIIm+JnsHoJ6GfTeNthQvzFRRmAiRXE82hTSqKP/Xk
# Wo79GwAW/WT1sY0asIZTtj+XRpCZ6mdxOYcIEL/s5SRQVjGO
# SIG # End signature block
```

Figure 11.7 – The signed file now contains a signature block

If we were to enable **ExecutionPolicy AllSigned** and attempt to run the self-signed script, we'd be asked if we really want to run software from this untrusted publisher:



```
PS C:\Windows\System32> Get-ExecutionPolicy
AllSigned
PS C:\Windows\System32> C:\tmp\HelloWorld.ps1

Do you want to run software from this untrusted publisher?
File C:\tmp\HelloWorld.ps1 is published by CN=Test Certificate and is not trusted on your system. Only run scripts from
 trusted publishers.
[V] Never run  [D] Do not run  [R] Run once  [A] Always run  [?] Help (default is "D"): r
Hello World!
PS C:\Windows\System32> _
```

Figure 11.8 – The ExecutionPolicy prompt

To execute this script, we must select **[R] Run once**. If you want to permanently run scripts from this publisher without being prompted each time, you can use the **[A] Always Run** option.

If you want to run scripts from this publisher without being prompted at all, you can add the self-signed certificate to the **Trusted Publishers** store. This allows you to establish a trusted relationship between the publisher and your computer, ensuring that scripts from the publisher are automatically trusted and executed without interruptions.

If we want to permanently run scripts from this publisher without being prompted, we need to add our self-signed certificate to the computer's **Trusted Publishers certificate store**:

```
> $publisherCertStore = [System.Security.Cryptography.X509Certificates.X509Store]::new("TrustedPublisher
```

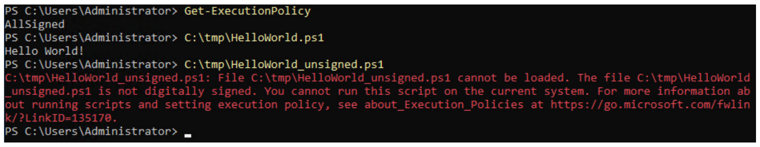
```
> $publisherCertStore.Open("ReadWrite")
```

```
> $publisherCertStore.Add($testCert)
```

```
> $publisherCertStore.Close()
```

By adding the certificate to the **Trusted Publishers** store, you can ensure that all the code signed by your self-signed certificate can be trusted. Since it is not possible to copy certificates from one store to another by using **Copy-Item**, we must use the **Certificate Store API** interface to access the **Trusted Publishers** certificate store, then open it with read/write permissions, add the certificate that we created earlier, and close the store again.

Now, if we execute the **HelloWorld.ps1** script again, it will run without prompting us, whereas an unsigned file would be rejected:

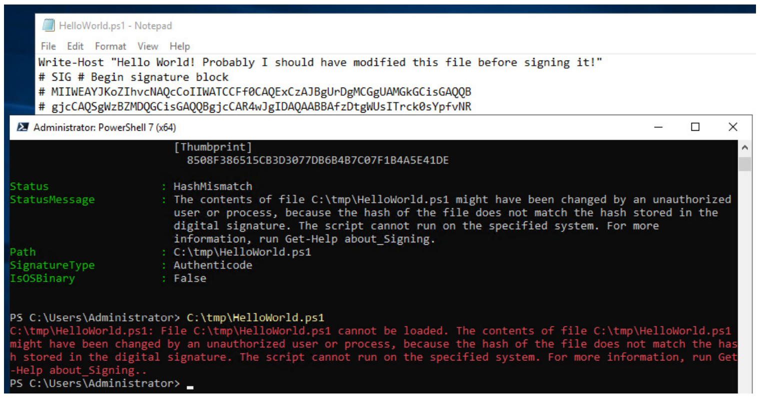


```
PS C:\Users\Administrator> Get-ExecutionPolicy
AllSigned
PS C:\Users\Administrator> C:\tmp\HelloWorld.ps1
Hello World!
PS C:\Users\Administrator> C:\tmp\HelloWorld_unsigned.ps1
C:\tmp\HelloWorld_unsigned.ps1: File C:\tmp\HelloWorld_unsigned.ps1 cannot be loaded. The file C:\tmp\HelloWorld
_unsigned.ps1 is not digitally signed. You cannot run this script on the current system. For more information ab
out running scripts and setting execution policy, see about_Execution_Policies at https://go.microsoft.com/fwlin
k/?LinkID=135170.
PS C:\Users\Administrator> _
```

Figure 11.9 – A signed file can be executed without any problems

If you have any application control mechanism in place, such as AppLocker or WDAP, only a signed file will be allowed to run – *if* the publisher was added as a trusted source for the application control mechanism to run. Depending on the application control system in use, this can be done using, for example, a **publisher rule** in a policy, or another similar mechanism to trust the publisher.

Since script signing adds a signature for exactly the file you signed, the file cannot be modified if the signature should remain valid. If you were to modify the content of the signed file and verify the signature using **Get-AuthenticodeSignature**, you would see that the hash of the signature does not match the content of the file anymore. Therefore, the signature will be invalid and the file cannot be executed any longer if protection mechanisms against unsigned scripts have been applied:



```
Write-Host "Hello World! Probably I should have modified this file before signing it!"
# SIG # Begin signature block
# MIIWEAYJKoZIhvcNAQcCoIIWATCCFf0CAQExCzAJBgUrDgMCGgUAMGkGCisGAQQB
# gjcCAQSgWzBZMDQGCisGAQQBgjcCAR4wJgIDAQAABBAfzDtgWUsITrck0sYpfvNR
```

```
                          [Thumbprint]
                            8508F386515CB3D3077DB6B4B7C07F1B4A5E41DE

Status                  : HashMismatch
StatusMessage           : The contents of file C:\tmp\HelloWorld.ps1 might have been changed by an unauthorized
                          user or process, because the hash of the file does not match the hash stored in the
                          digital signature. The script cannot run on the specified system. For more
                          information, run Get-Help about_Signing.
Path                    : C:\tmp\HelloWorld.ps1
SignatureType           : Authenticode
IsOSBinary              : False


PS C:\Users\Administrator> C:\tmp\HelloWorld.ps1
C:\tmp\HelloWorld.ps1: File C:\tmp\HelloWorld.ps1 cannot be loaded. The contents of file C:\tmp\HelloWorld.ps1
might have been changed by an unauthorized user or process, because the hash of the file does not match the has
h stored in the digital signature. The script cannot run on the specified system. For more information, run Get
-Help about_Signing..
PS C:\Users\Administrator> _
```

Figure 11.10 – HashMismatch after changing the signed file's content

Therefore, whenever you modify the content of a signed file, you will need to sign it once more. If you have a **continuous integration/continuous delivery** (**CI/CD**) pipeline in place, script signing can easily be automated using the **Set-AuthenticodeSignature** cmdlet.

There are several ways to build a CI/CD pipeline if you are new to this concept. Just to mention a few, a CI/CD pipeline can, for example, be realized using Azure DevOps or GitHub.

The following are some resources to help you get started with this:

- *Design a CI/CD pipeline using Azure DevOps*: **https://learn.microsoft.com/en-us/azure/devops/pipelines/architectures/devops-pipelines-baseline-architecture**
- *How to build a CI/CD pipeline with GitHub Actions in four simple steps*: **https://github.blog/2022-02-02-build-ci-cd-pipeline-github-actions-four-steps/**

It's important to also make sure you apply code signing best practices when you are planning to use code signing in your production environment. Microsoft has published a *Code Signing Best Practices* document for this, which you use as a reference: http://download.microsoft.com/download/a/f/7/af7777e5-7dcd-4800-8a0a-b18336565f5b/best_practices.doc.

Code signing is a great way to ensure that your scripts are legit and were not tampered with. But as you learned earlier in this book, the execution policy alone is not a security boundary and can easily be bypassed. Therefore, only relying on the execution policy is not a good idea. If you want to prevent unauthorized scripts from running in your environment, you need to implement an application control solution.

Controlling applications and scripts

An application control solution not only protects against unauthorized PowerShell scripts; it can also be used to define which applications, executables, and DLLs are allowed to run in the environment.

It is important to keep in mind that while PowerShell attacks may seem like a concern for many professionals, they represent a relatively small portion of the malware that makes its way onto systems. It is essential to not overlook the danger posed by traditional executable and DLL attacks.

Application control solutions often provide a possibility to also just prohibit single unwanted applications, but the desired outcome should always be to prohibit everything and configure all allowed applications. As you may recall from ***Chapter 5***, *PowerShell Is Powerful – System and API Access*, even if you block **PowerShell.exe** in your environment, it is still possible to run it by just using the native API functions, irrespective of whether it makes sense to block PowerShell (you shouldn't, of course; it's better to implement and leverage a proper logging and security strategy instead).

If you were to only prohibit unwanted applications, attackers would always find a way to circumvent your restrictions – there's just too much to block and only prohibiting unwanted applications would make your environment always vulnerable to attacks.

It's better to directly start by auditing what software is used and needed in your environment, implementing a proper application control strategy, and preventing everything else from being run.

There are many application control tools on the market, but in this book, we will only look at Microsoft AppLocker and WDAC.

Planning for application control

Before applying strict rules to enforce application control to your production environment, make sure that you always audit and create a software catalog of the applications used. You don't want to impact your employees in such a way that they are no longer able to work.

Even if you are only implementing an audit policy, you have already significantly improved the signal-to-noise ratio in your SIEM. Consider this scenario: before implementing application control, your SIEM is flooded with thousands of events every day from known and authorized applications, making it extremely challenging to identify potential malware or unwanted software.

But if you are only able to implement 80% of an application control policy, and therefore only enable auditing, the number of events already decreases to a manageable level. In this case, you would be left with only a few hundred events per day, which contain legitimate software operations and a potential subset of unwanted software or malware. This approach already reduces the noise in your SIEM significantly and enables you to defend your environment in a much better way.

Once you have created the first policy, make sure you test it before rolling it out. Once you are ready to deploy it, follow the following rollout strategy:

1. Test your policy in a test environment.
2. It can be very useful to announce your configuration changes as early as possible so that your employees can plan better.
3. Divide your tech department into several groups, then slowly roll out the policy for the first group, review audit logs, and fix problems on the fly. Once fixed, roll out the policy to the next group and so on.
4. If everything worked during the last deployment step, enroll your policy for power users in your environment. Needless to say, always communicate it to the people who'd be affected before rolling out such a policy.
5. After fixing all probable configuration issues, slowly roll out the policy department by department. Always make sure you divide each group into sub-groups and communicate it to the affected employees before enforcing changes.

Always review your blocked applications regularly. This not only helps you identify problems your users might have but also helps you spot the beginning of an attack.

It takes some time to identify which applications are in use and to adjust your configuration accordingly, but it is worth the effort and it will help you harden your environment enormously.

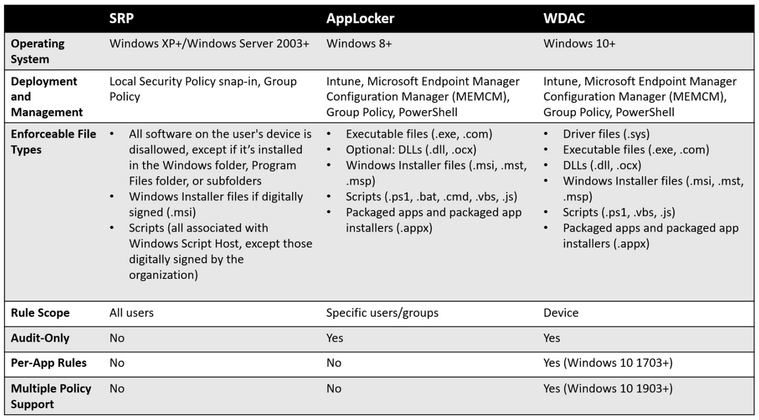
First, let's look at which application control options are available on Windows operating systems.

Built-in application control solutions

Over the years, Microsoft has worked on several solutions for application control, starting with SRP with Windows XP to AppLocker, which was introduced with Windows 8 – until they finally released WDAC with Windows 10.

Over the years, capabilities have been improved enormously and each tool brought advantages to their former versions. If possible, always use WDAC for application control as it will be continuously improved. But if you are still using older operating system versions that you need to restrict, it is possible to run all three solutions in parallel.

The following figure provides you with a simplified comparison of all three solutions:



	SRP	AppLocker	WDAC
Operating System	Windows XP+/Windows Server 2003+	Windows 8+	Windows 10+
Deployment and Management	Local Security Policy snap-in, Group Policy	Intune, Microsoft Endpoint Manager Configuration Manager (MEMCM), Group Policy, PowerShell	Intune, Microsoft Endpoint Manager Configuration Manager (MEMCM), Group Policy, PowerShell
Enforceable File Types	• All software on the user's device is disallowed, except if it's installed in the Windows folder, Program Files folder, or subfolders • Windows Installer files if digitally signed (.msi) • Scripts (all associated with Windows Script Host, except those digitally signed by the organization)	• Executable files (.exe, .com) • Optional: DLLs (.dll, .ocx) • Windows Installer files (.msi, .mst, .msp) • Scripts (.ps1, .bat, .cmd, .vbs, .js) • Packaged apps and packaged app installers (.appx)	• Driver files (.sys) • Executable files (.exe, .com) • DLLs (.dll, .ocx) • Windows Installer files (.msi, .mst, .msp) • Scripts (.ps1, .vbs, .js) • Packaged apps and packaged app installers (.appx)
Rule Scope	All users	Specific users/groups	Device
Audit-Only	No	Yes	Yes
Per-App Rules	No	No	Yes (Windows 10 1703+)
Multiple Policy Support	No	No	Yes (Windows 10 1903+)

Figure 11.11 – Simplified comparison of SRP, AppLocker, and WDAC

Of course, this is not a complete list of all features. Please refer to the following links for a more detailed overview of which differences exist between SRP, AppLocker, and WDAC:

- *What features are different between Software Restriction Policies and AppLocker?*: **https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/what-is-applocker#what-features-are-different-between-software-restriction-policies-and-applocker**
- *Windows Defender Application Control and AppLocker feature availability*: **https://learn.microsoft.com/en-us/windows/security/application-security/application-control/windows-defender-application-control/feature-availability**

These solutions are huge topics, so you will only find an overview of each technology, as well as some tips and tricks that will help you start implementing your own application control rules. As the focus of this book is PowerShell, we will also focus mostly on restricting and using PowerShell in this chapter.

Getting familiar with Microsoft AppLocker

AppLocker is Microsoft's successor to SRP and was introduced with Windows 7. You can use it to extend SRP's function, as well as its features.

In comparison to SRP, AppLocker policies can be scoped to specific users or groups and it's also possible to audit before you enforce rules. It is possible to deploy SRP and AppLocker policies in parallel in various ways; take a look at the following documentation:

- *Use AppLocker and Software Restriction Policies in the same domain*: **https://learn.microsoft.com/en-us/windows/security/application-security/application-control/windows-defender-application-control/applocker/use-applocker-and-software-restriction-policies-in-the-same-domain**
- *Use Software Restriction Policies and AppLocker policies*: **https://learn.microsoft.com/en-us/windows/security/application-security/application-control/windows-defender-application-control/applocker/using-software-restriction-policies-and-applocker-policies**

Computers on which you want to deploy AppLocker need to have an operating system installed that allows AppLocker policies to be enforced, such as Windows Enterprise. You can also create AppLocker rules on a computer running Windows Professional. However, it is only possible to enforce AppLocker rules on Windows Professional and other operating system versions if they are managed with Intune. If AppLocker rules are not enforced, they don't apply and give you no protection at all.

If you want to restrict applications on unsupported operating systems, you can either deploy SRP rules in parallel or use WDAC.

For AppLocker to work properly, it is required that the **Application Identity** service is running.

Deploying AppLocker

You can deploy AppLocker using GPO, Intune, **Microsoft Configuration Manager**, and PowerShell. Of course, you can also use Local Group Policy Editor for testing purposes. However, it is not possible to enforce AppLocker rules using this method, so you should avoid it in production.

When working with AppLocker, there are five different rule types that you can configure:

- **Executable Rules**: Using **Executable Rules**, you can restrict executables that end in **.exe** and **.com**.
- **Windows Installer Rules**: By configuring **Windows Installer Rules**, you can restrict **.msi**, **.mst**, and **.msp** Windows Installer files.
- **Script Rules**: With **Script Rules**, you can restrict **.ps1**, **.bat**, **.cmd**, **.vbs**, and **.js** script files.
- **DLL rules**: You can use DLL rules to restrict **.dll** and **.ocx** files.

Although DLL rules were once considered optional due to concerns about performance, in today's security landscape, an app control system without DLL enforcement enabled is incomplete and leaves your environment vulnerable. These rules have to be enabled before they can be used and configured using GPO or a local Group Policy. If you are using GPOs for your configuration, go to **Computer Configuration** | **Policies** | **Windows Settings** | **Security Settings** | **Application Control Policies** | **AppLocker**. Then, right-click **AppLocker** and select **Properties** | **Advanced** | **Enable the DLL rule collection.**

- **Packaged app Rules**: Using **Packaged app Rules**, you can restrict **.appx** package files.

For every rule you create, you need to select an action. Here, you must decide whether a file should be allowed or blocked by choosing either **Allow** or **Deny**. Usually, you want to block everything and only allow the selected applications.

Using AppLocker rules, it is also possible to scope the rule to a particular **User or group**. If nothing is specified in particular, the rule applies to **Everyone**.

You will also need to decide on the *primary condition* that the rule should contain. For **Packaged app Rules**, you can only configure a **Publisher** condition; for all other rules, **Path** and **File hash** conditions can be applied – in addition to the **Publisher** conditions:

- **Path**: Using the **Path** condition, you can specify a path that will be either allowed or denied by your rule. You can also define an exception.

Using the **Path** condition is the most insecure condition as file and path names can easily be changed to bypass your rules. If possible, try to avoid path rules.

- **Publisher**: When using the **Publisher** condition, a file needs to be digitally signed. Using this condition, you can not only specify the publisher – you can also specify the product name, the filename, as well as the file version that a file should have to be allowed or denied. It is also possible to define exceptions.
- **File hash**: A cryptographic file hash will be calculated for this file. If the file changes, the file hash will change as well. Therefore, a hash can only apply to one file and you need to configure a file hash condition for every file you want to allow or deny if this condition is used.

All these rules, actions, user scopes, and conditions apply to all configuration methods.

Configuring AppLocker in your environment can take some time, but it is worth it once you have implemented it. To help you with your initial configuration, Aaron Margosis released *AaronLocker* on GitHub: **https://github.com/microsoft/AaronLocker**.

This script and documentation collection should help make your initial configuration, as well as the maintenance of your AppLocker rules, as easy as possible.

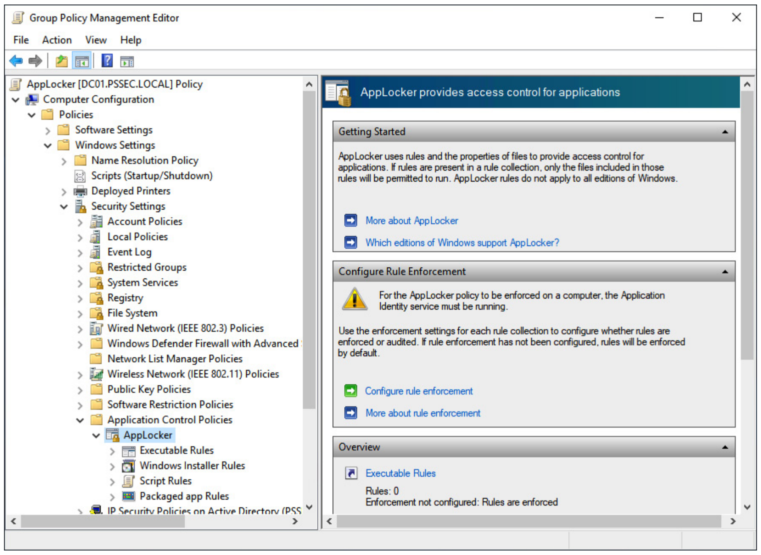
Behind AaronLocker – Where Did the Name Come From?

The name *AaronLocker* was not Aaron's idea himself – it was the idea of my friend and long-time mentor Chris Jackson, who unfortunately passed away some time ago (rest in peace, Chris!). Aaron was not especially fond to call his product after his first name, but since he could not think of a better name, he gave in to Chris' idea and so the name *AaronLocker* was born.

However, we have only learned what AppLocker rules consist of and not how to deploy and configure them using different deployment methods. Therefore, as a next step, we'll explore how AppLocker can be managed.

GPO

If you are using GPOs or Local Group Policy for your configuration, navigate to **Computer Configuration** | **Policies** | **Windows Settings** | **Security Settings** | **Application Control Policies** | **AppLocker**. In this section, you will find the **Executable Rules**, **Windows Installer Rules**, **Script Rules**, and **Packaged app Rules** options, as shown here:


Group Policy Management Editor

File Action View Help

```
AppLocker [DC01.PSSEC.LOCAL] Policy
  Computer Configuration
    Policies
      Software Settings
      Windows Settings
        Name Resolution Policy
        Scripts (Startup/Shutdown)
        Deployed Printers
        Security Settings
          Account Policies
          Local Policies
          Event Log
          Restricted Groups
          System Services
          Registry
          File System
          Wired Network (IEEE 802.3) Policies
          Windows Defender Firewall with Advanced
          Network List Manager Policies
          Wireless Network (IEEE 802.11) Policies
          Public Key Policies
          Software Restriction Policies
          Application Control Policies
            AppLocker
              Executable Rules
              Windows Installer Rules
              Script Rules
              Packaged app Rules
          IP Security Policies on Active Directory (PSS
```

AppLocker provides access control for applications

Getting Started

AppLocker uses rules and the properties of files to provide access control for applications. If rules are present in a rule collection, only the files included in those rules will be permitted to run. AppLocker rules do not apply to all editions of Windows.

More about AppLocker

Which editions of Windows support AppLocker?

Configure Rule Enforcement

For the AppLocker policy to be enforced on a computer, the Application Identity service must be running.

Use the enforcement settings for each rule collection to configure whether rules are enforced or audited. If rule enforcement has not been configured, rules will be enforced by default.

Configure rule enforcement

More about rule enforcement

Overview

Executable Rules

Rules: 0
Enforcement not configured: Rules are enforced

Figure 11.12 – Configuring AppLocker using GPO

To enable the enforcement or auditing behavior, right-click on AppLocker and select **Properties**. In the window that appears, you can configure which AppLocker rules should be enforced or audited.

If you are using GPOs as a configuration method, make sure that all the systems you want to configure have at least Windows 10 Enterprise installed. Otherwise, you cannot enforce AppLocker rules.

If you also want to enable DLL rules, you can do this by right-clicking on **AppLocker** and selecting **Properties** | **Advanced** | **Enable the DLL rule collection**. Refer to the descriptions of the DLL rules to learn more about them. After enabling DLL rules, they will show up under AppLocker.

Intune

Before you can configure AppLocker via Intune, you will need to create an AppLocker policy using GPO or Local Group Policy. Once your configuration is ready, export it by right-clicking on **AppLocker** and selecting **Export Policy**:

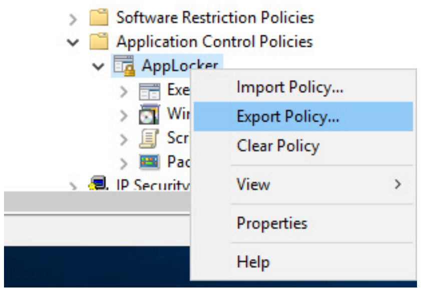
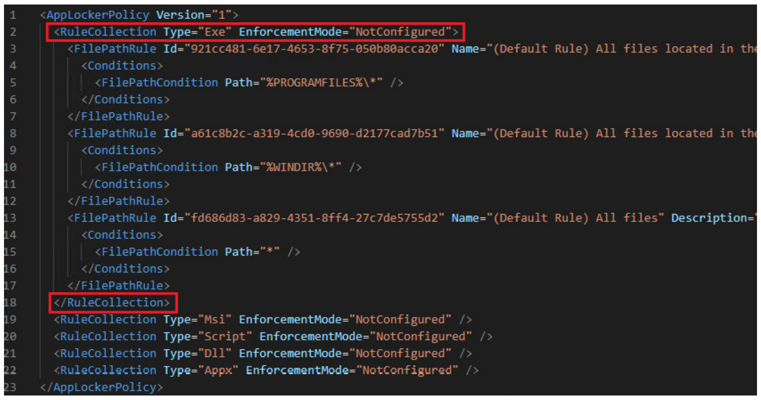


Figure 11.13 – Exporting the AppLocker policy

A window will appear where you need to select where your exported policy should be saved to. Select a path and confirm it; your AppLocker policy will be successfully exported as a **.xml** file.

Unfortunately, you cannot just copy and paste the content of the file into your Intune configuration. Therefore, open the file with an editor and search for each rule type for its section. This is indicated by the **<RuleCollection ...> ... </RuleCollection>** tags from **RuleCollection**.

There's one **RuleCollection** section for every rule type, so if you want to get the **RuleCollection** section for all executable files, select everything between **<RuleCollection Type="Exe" EnforcementMode="NotConfigured">**, including the surrounding tags, as shown in the following screenshot. If needed, repeat this for the other available rule types:



```
<AppLockerPolicy Version="1">
  <RuleCollection Type="Exe" EnforcementMode="NotConfigured">
    <FilePathRule Id="921cc481-6e17-4653-8f75-050b80acca20" Name="(Default Rule) All files located in the
      <Conditions>
        <FilePathCondition Path="%PROGRAMFILES%\*" />
      </Conditions>
    </FilePathRule>
    <FilePathRule Id="a61c8b2c-a319-4cd0-9690-d2177cad7b51" Name="(Default Rule) All files located in the
      <Conditions>
        <FilePathCondition Path="%WINDIR%\*" />
      </Conditions>
    </FilePathRule>
    <FilePathRule Id="fd686d83-a829-4351-8ff4-27c7de5755d2" Name="(Default Rule) All files" Description="
      <Conditions>
        <FilePathCondition Path="*" />
      </Conditions>
    </FilePathRule>
  </RuleCollection>
  <RuleCollection Type="Msi" EnforcementMode="NotConfigured" />
  <RuleCollection Type="Script" EnforcementMode="NotConfigured" />
  <RuleCollection Type="Dll" EnforcementMode="NotConfigured" />
  <RuleCollection Type="Appx" EnforcementMode="NotConfigured" />
</AppLockerPolicy>
```

Figure 11.14 – Selecting the RuleCollection section for executable rules

Configuring AppLocker using Intune relies on the AppLocker **configuration service provider** (**CSP**): https://docs.microsoft.com/en-us/windows/client-management/mdm/applocker-csp.

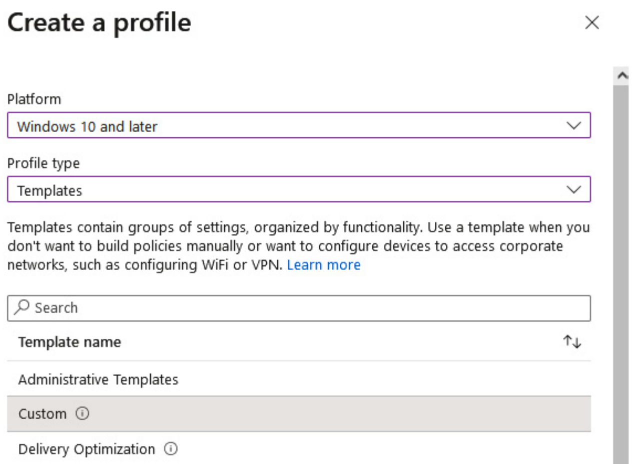
The CSP provides an interface that allows **mobile device management** (**MDM**) solutions to control, configure, read, delete, and edit the configuration settings of the device that's being managed. A custom configuration for a Windows 10 device can be configured using the **Open Mobile Alliance Uniform Resource Identifier** (**OMA-URI**) string.

Thanks to Intune and the AppLocker CSP, most operating systems can be configured to use AppLocker in Enforcement mode:

- *Configuration Service Provider*: https://docs.microsoft.com/en-us/windows/client-management/mdm/configuration-service-provider-reference#csp-support
- *Deploy OMA-URIs to target a CSP through Intune, and a comparison to on-premises*: https://learn.microsoft.com/en-us/troubleshoot/mem/intune/device-configuration/deploy-oma-uris-to-target-csp-via-intune

Now, in Intune, go to **Devices** | **Configuration Profiles** and click on **Create Profile**.

Select **Windows 10 and Later** under **Platform**, **Templates** under **Profile Type**, and **Custom** under **Template**, then click **Create**:


Create a profile

Platform: Windows 10 and later

Profile type: Templates

Templates contain groups of settings, organized by functionality. Use a template when you don't want to build policies manually or want to configure devices to access corporate networks, such as configuring WiFi or VPN. Learn more

Figure 11.15 – Create a profile

On the next page, name your AppLocker policy – for example, **AppLocker Policy** – and click **Next**.

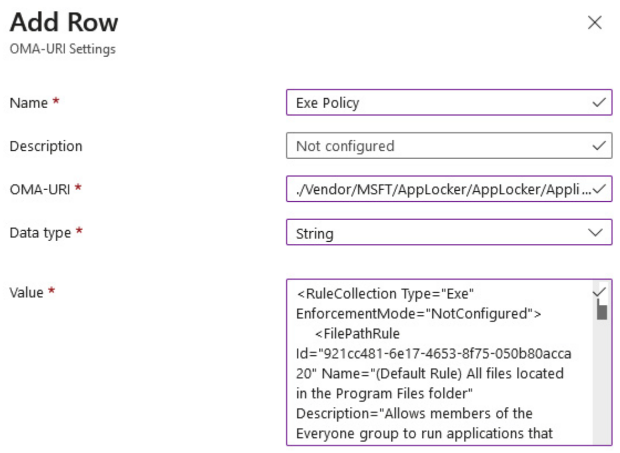
In the **OMA-URI Settings** section, select **Add** to add your AppLocker rule configuration. This is where you create the actual policy, using the snippet from your **.xml** export.

First, type a name that represents the policy well, such as **Exe Policy**, if you want to start configuring the policy for **.exe** files in your environment.

In the **OMA-URI** field, type the string according to the policy you are just configuring:

- **Exe**: **./Vendor/MSFT/AppLocker/AppLocker/ApplicationLaunchRestrictions/apps/EXE/Policy**
- **MSI**: **./Vendor/MSFT/AppLocker/ApplicationLaunchRestrictions/apps/MSI/Policy**
- **Script**: **./Vendor/MSFT/AppLocker/ApplicationLaunchRestrictions/apps/Script/Policy**
- **DLL**: **./Vendor/MSFT/AppLocker/ApplicationLaunchRestrictions/apps/DLL/Policy**
- **Appx**: **./Vendor/MSFT/AppLocker/ApplicationLaunchRestrictions/apps/StoreApps/Policy**

Change **Data type** to **String** and paste the **RuleCollection** lines that you copied earlier from the exported **.xml** file. Click **Save**. Add a policy using the **OMA-URI Settings** area for every rule type you want to configure. Once you are finished, click **Review + save** to save your configuration:



Name * Exe Policy
Description Not configured
OMA-URI * ./Vendor/MSFT/AppLocker/AppLocker/Appli...
Data type * String
Value *
```
<RuleCollection Type="Exe"
EnforcementMode="NotConfigured">
    <FilePathRule
Id="921cc481-6e17-4653-8f75-050b80acca
20" Name="(Default Rule) All files located
in the Program Files folder"
Description="Allows members of the
Everyone group to run applications that
```

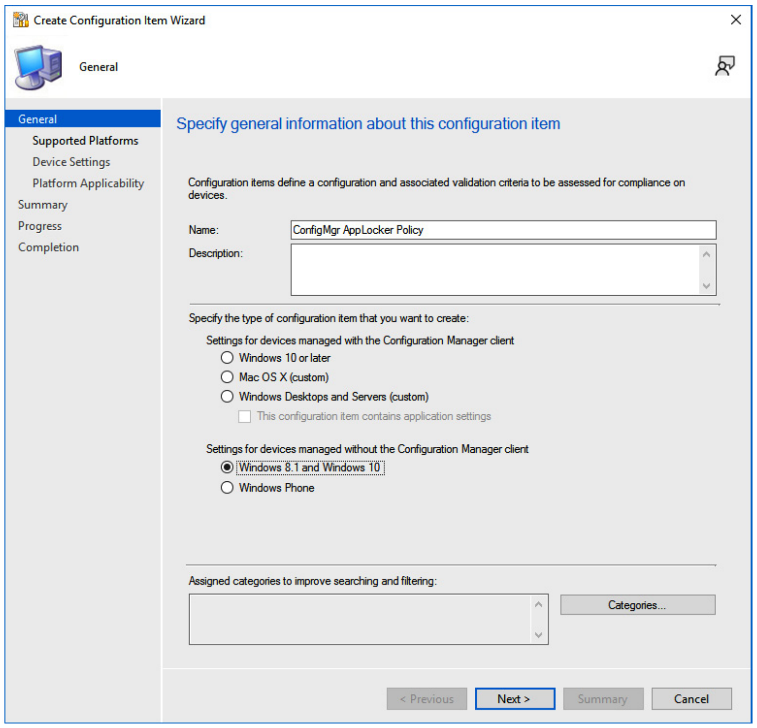
Figure 11.16 – Configuring the OMA-URI settings

As a next step, you can add computer groups to which these rules should apply. Click **Next** until you are in the **Review** + **create** section and review your rules. If everything seems fine, click **Create** to create your AppLocker rules.

Microsoft Configuration Manager

Configuration Manager was formerly known as **System Center Configuration Manager** (**SCCM**). Configuration Manager contains a lot of preconfigured configuration options and packages, but unfortunately, there is no preconfigured option for AppLocker. However, it still can be deployed using custom configuration options.

Under **Compliance Settings**, create a new **Configuration Item**; in the **Create Configuration Item Wizard** area, specify a name for your new policy and select **Windows 8.1 and Windows 10** under **Settings for devices managed without the Configuration Manager client**:



Specify general information about this configuration item

Configuration items define a configuration and associated validation criteria to be assessed for compliance on devices.

Name: ConfigMgr AppLocker Policy
Description:

Specify the type of configuration item that you want to create:

Settings for devices managed with the Configuration Manager client
- ○ Windows 10 or later
- ○ Mac OS X (custom)
- ○ Windows Desktops and Servers (custom)
  - ☐ This configuration item contains application settings

Settings for devices managed without the Configuration Manager client
- [✓ Windows 8.1 and Windows 10]
- ○ Windows Phone

Assigned categories to improve searching and filtering:

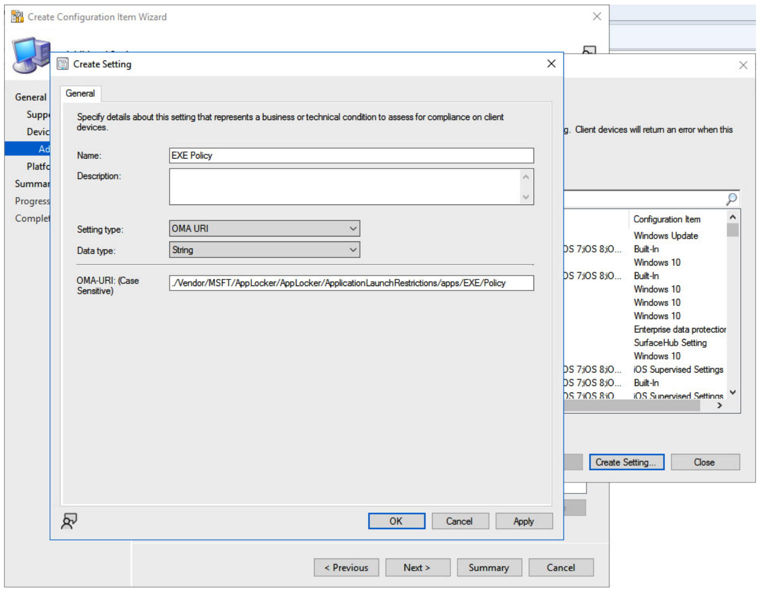
Figure 11.17 – Creating a custom AppLocker policy using Configuration Manager

Similar to the configuration with Intune, we can also use AppLocker CSP for the configuration with Configuration Manager.

Next, select for which platforms you want to configure AppLocker – in my example, I chose **Windows 10** only and clicked **Next**.

As a next step, don't select any device settings; instead, check the **Configure additional settings that are not in the default setting groups** checkbox and click **Next**.

In the **Additional Settings** pane, click **Add**. The **Browse Settings** window will open. Now, click **Create Setting…**. A new window called **Create Setting** will open, as shown here:



Specify details about this setting that represents a business or technical condition to assess for compliance on client devices.

Name: EXE Policy

Description:

Setting type: OMA URI

Data type: String

OMA-URI: (Case Sensitive) ./Vendor/MSFT/AppLocker/AppLocker/ApplicationLaunchRestrictions/apps/EXE/Policy

g. Client devices will return an error when this

Figure 11.18 – Specifying the policy's name and the OMA-URI

In the **Create Setting** dialog, enter the setting's **Name** and specify the string of the OMA-URI, as we did in the *Intune configuration* section (this is also where you can find the summarized OMA-URI strings in this book). Click **OK**.

As a next step, specify the rules for this setting by double clicking the setting that you just created and enter a meaningful **Name**, select **Value** under **Rule type**, and ensure **EXE Policy** (or the setting name that you configured earlier) **Equals** the *RuleCollection XML snippet* that we created earlier in the **Intune** section.

Usually, Configuration Manager items are used to query a state. If the state is different from the desired outcome, you can optionally configure the rule to be remediated automatically by checking the **Remediate non-compliant rules when supported** option.

Repeat this step for every rule type that you want to configure until all the rules are configured accordingly.

Click **Next** until **Create Configuration Item Wizard task** shows up as completed successfully.

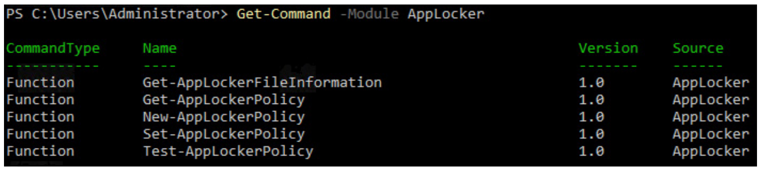
Now, create a **Configuration Baseline** task, enter a meaningful name, and click **Add**. Select the formerly created policy to be added to this baseline and confirm this with **OK**.

Last, but not least, **Deploy** the new configuration baseline by selecting the baseline and configuring a **compliance evaluation schedule** to define in which interval the baseline is checked and applied. In my case, I have stated that this baseline should be run daily. Again, confirm this with **OK**.

PowerShell

Of course, you can also use PowerShell to configure and read AppLocker rules. You can use the module AppLocker for this, which already contains several functions to help you with this job.

The following screenshot provides an overview of all AppLocker-related PowerShell commands:

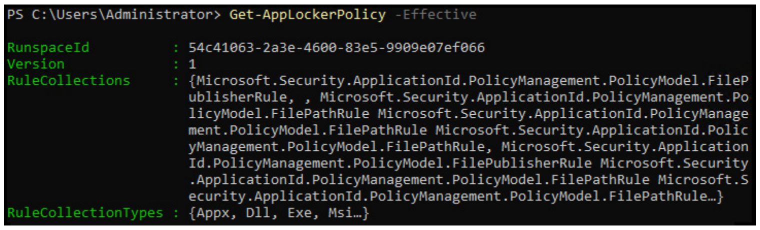


```
PS C:\Users\Administrator> Get-Command -Module AppLocker

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Function        Get-AppLockerFileInformation                       1.0        AppLocker
Function        Get-AppLockerPolicy                                1.0        AppLocker
Function        New-AppLockerPolicy                                1.0        AppLocker
Function        Set-AppLockerPolicy                                1.0        AppLocker
Function        Test-AppLockerPolicy                               1.0        AppLocker
```

Figure 11.19 – Functions within the AppLocker module

At first glance, it looks like the module provides very limited functionality, but let's look deeper into each function; they have way more functionality than you would expect and allow you to work even more efficiently than with the user interface.

Get-AppLockerPolicy helps you find out if there is an AppLocker policy in place. Using the **-Effective** parameter, you can see if a policy has been specified at all:

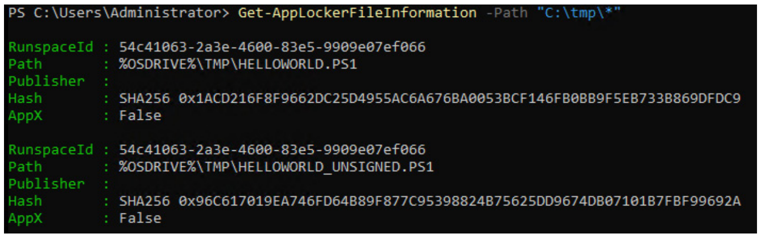


```
PS C:\Users\Administrator> Get-AppLockerPolicy -Effective

RunspaceId         : 54c41063-2a3e-4600-83e5-9909e07ef066
Version            : 1
RuleCollections    : {Microsoft.Security.ApplicationId.PolicyManagement.PolicyModel.FileP
                     ublisherRule, , Microsoft.Security.ApplicationId.PolicyManagement.Po
                     licyModel.FilePathRule Microsoft.Security.ApplicationId.PolicyManage
                     ment.PolicyModel.FilePathRule Microsoft.Security.ApplicationId.Polic
                     yManagement.PolicyModel.FilePathRule, Microsoft.Security.Application
                     Id.PolicyManagement.PolicyModel.FilePublisherRule Microsoft.Security
                     .ApplicationId.PolicyManagement.PolicyModel.FilePathRule Microsoft.S
                     ecurity.ApplicationId.PolicyManagement.PolicyModel.FilePathRule…}
RuleCollectionTypes : {Appx, Dll, Exe, Msi…}
```

Figure 11.20 – Getting the effective AppLocker policy using the Get-AppLocker policy

You can also use the **-Local** parameter to see what is defined in the local AppLocker policy. The **-Domain** parameter, combined with the **-Ldap** parameter, helps you see the current domain-configured AppLocker policy. And of course, you can also investigate a policy out of a **.xml** file using the **-Xml** parameter.

Using **Get-AppLockerFileInformation** allows you to get all the information from either a file, a path, or an event log:



```
PS C:\Users\Administrator> Get-AppLockerFileInformation -Path "C:\tmp\*"

RunspaceId : 54c41063-2a3e-4600-83e5-9909e07ef066
Path       : %OSDRIVE%\TMP\HELLOWORLD.PS1
Publisher  :
Hash       : SHA256 0x1ACD216F8F9662DC25D4955AC6A676BA0053BCF146FB0BB9F5EB733B869DFDC9
AppX       : False

RunspaceId : 54c41063-2a3e-4600-83e5-9909e07ef066
Path       : %OSDRIVE%\TMP\HELLOWORLD_UNSIGNED.PS1
Publisher  :
Hash       : SHA256 0x96C617019EA746FD64B89F877C95398824B75625DD9674DB07101B7FBF99692A
AppX       : False
```

Figure 11.21 – Retrieving AppLocker file information using Get-AppLockerFileInformation

In the preceding screenshot, you can see the AppLocker information of both demo scripts from our code signing example earlier. Usually, if the script had been signed by a corporate or public CA, you would also see the publisher information, but since we used a self-signed script, which is only meant for testing purposes, this certificate has no publisher and therefore we cannot use it to create an AppLocker publisher rule.

Usually, the most common way to generate AppLocker rules is by creating a policy based on a **golden image** of a server or client system, instead of manually selecting individual files and directories. To do this, you can use the **Get-AppLockerFileInformation** cmdlet to identify all the files that are authorized to run on the image and then use the **New-AppLockerPolicy** cmdlet to automatically generate the corresponding AppLocker rules for each file.

The following example takes all the files in the **C:\** drive and generates a rule for each – the resulting file will be saved under **C:\tmp\Applocker.xml**:

```
> Get-AppLockerFileInformation -Directory 'C:\' -Recurse -ErrorAction SilentlyContinue | New-AppLockerPo
```

Once the file has been created, you will need to test and fine-grain it to deploy AppLocker rules for your golden image.

Another very effective way to deploy AppLocker is to capture events from existing *known good* systems that have the required software installed and are considered uncompromised. Using those events to generate a policy with PowerShell can save you a lot of time and effort. It is even possible to pipe in file information from event logs to automatically generate AppLocker rules. This can be especially useful when dealing with large and complex environments where manually creating rules can be a daunting task:

```
> Get-AppLockerFileInformation -EventLog -EventType Audited | New-AppLockerPolicy -RuleType Publisher,Ha
```

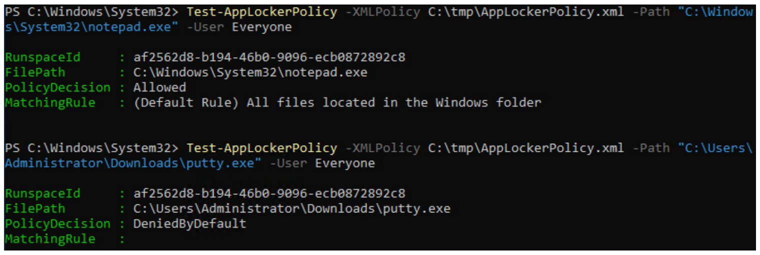
You can then use the **Set-AppLockerPolicy** cmdlet to configure Group Policy or Local Group Policy with the specified AppLocker configuration:

```
Set-AppLockerPolicy -XmlPolicy "C:\tmp\AppLockerPolicy.xml"
```

To configure GPO on a remote domain controller, make sure you use the **-Ldap** parameter and configure the LDAP path to where the policy is located. If you want to merge the existing policy with a newly configured one, make sure you specify the **-Merge** parameter.

This cmdlet only works with Group Policy or local policy. If you have AppLocker configured via AppLocker CSP, this cmdlet won't work.

Using the **Test-AppLockerPolicy** cmdlet, you can test your AppLocker policy to find out if a certain file would be allowed to be executed if the specified policy were to apply:



```
PS C:\Windows\System32> Test-AppLockerPolicy -XMLPolicy C:\tmp\AppLockerPolicy.xml -Path "C:\Window
s\System32\notepad.exe" -User Everyone

RunspaceId     : af2562d8-b194-46b0-9096-ecb0872892c8
FilePath       : C:\Windows\System32\notepad.exe
PolicyDecision : Allowed
MatchingRule   : (Default Rule) All files located in the Windows folder


PS C:\Windows\System32> Test-AppLockerPolicy -XMLPolicy C:\tmp\AppLockerPolicy.xml -Path "C:\Users\
Administrator\Downloads\putty.exe" -User Everyone

RunspaceId     : af2562d8-b194-46b0-9096-ecb0872892c8
FilePath       : C:\Users\Administrator\Downloads\putty.exe
PolicyDecision : DeniedByDefault
MatchingRule   :
```

Figure 11.22 – Using Test-AppLockerPolicy to find out whether notepad.exe or putty.exe would be allowed to run

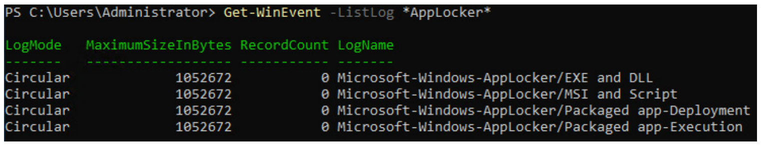
In this screenshot, you can see that using this AppLocker policy, **notepad.exe** would be allowed to run, while **putty.exe** would be prohibited as no matching allow rule has been configured.

Before you start deploying AppLocker in Enforce Rules Enforcement mode, you will want to audit what applications and scripts can be used in your environment regularly using **Audit only Enforcement** mode. This will let you allowlist them before you enforce your rules. You can do this using the logging capability by reviewing event logs.

Audit AppLocker events

When using event logs, you can not only find out which applications would have been blocked when using **Audit only Enforcement** mode – you can also find a lot more interesting information on how your AppLocker policies were applied or what applications did run in **Enforce Rules Enforcement** mode.

Using PowerShell, you can quickly get an overview of all AppLocker-related event logs by running **Get-WinEvent -ListLog *AppLocker***:



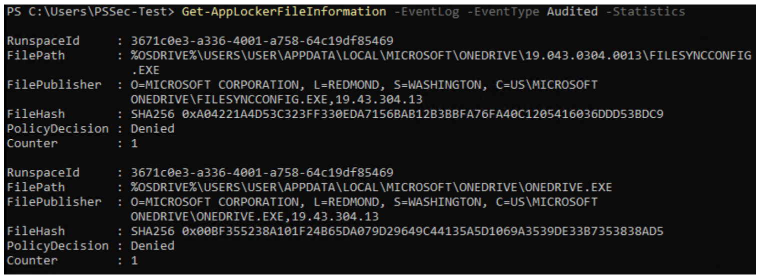
```
PS C:\Users\Administrator> Get-WinEvent -ListLog *AppLocker*

LogMode   MaximumSizeInBytes RecordCount LogName
-------   ------------------ ----------- -------
Circular             1052672           0 Microsoft-Windows-AppLocker/EXE and DLL
Circular             1052672           0 Microsoft-Windows-AppLocker/MSI and Script
Circular             1052672           0 Microsoft-Windows-AppLocker/Packaged app-Deployment
Circular             1052672           0 Microsoft-Windows-AppLocker/Packaged app-Execution
```

Figure 11.23 – AppLocker event logs

To get all the event IDs from a particular log, use **Get-WinEvent**, followed by the name of the event log. If you want to get all event IDs from the **Microsoft-Windows-AppLocker/EXE and DLL** log, for example, you can run **Get-WinEvent "Microsoft-Windows-AppLocker/EXE and DLL"**.

You can find more detailed information on AppLocker event logs and all event IDs in **Chapter 4**, *Detection – Auditing and Monitoring*.

To plan for your AppLocker deployment, it can be also very useful to review the statistics of what applications were allowed, denied, or audited. You can achieve this using **Get-AppLockerFileInformation**, as shown in the following screenshot:



```
PS C:\Users\PSSec-Test> Get-AppLockerFileInformation -EventLog -EventType Audited -Statistics

RunspaceId     : 3671c0e3-a336-4001-a758-64c19df85469
FilePath       : %OSDRIVE%\USERS\USER\APPDATA\LOCAL\MICROSOFT\ONEDRIVE\19.043.0304.0013\FILESYNCCONFIG
                 .EXE
FilePublisher  : O=MICROSOFT CORPORATION, L=REDMOND, S=WASHINGTON, C=US\MICROSOFT
                 ONEDRIVE\FILESYNCCONFIG.EXE,19.43.304.13
FileHash       : SHA256 0xA04221A4D53C323FF330EDA7156BAB12B3BBFA76FA40C1205416036DDD53BDC9
PolicyDecision : Denied
Counter        : 1

RunspaceId     : 3671c0e3-a336-4001-a758-64c19df85469
FilePath       : %OSDRIVE%\USERS\USER\APPDATA\LOCAL\MICROSOFT\ONEDRIVE\ONEDRIVE.EXE
FilePublisher  : O=MICROSOFT CORPORATION, L=REDMOND, S=WASHINGTON, C=US\MICROSOFT
                 ONEDRIVE\ONEDRIVE.EXE,19.43.304.13
FileHash       : SHA256 0x00BF355238A101F24B65DA079D29649C44135A5D1069A3539DE33B7353838AD5
PolicyDecision : Denied
Counter        : 1
```

Figure 11.24 – Reviewing the statistics of audited applications

Using **EventType**, you can choose between **Allowed**, **Denied**, or **Audited**. By doing this, you can see all the information about the file, as well as how often it tried to run the application and the decision of whether a file was or would have been allowed or denied.

Please refer to the following link to learn more about how to monitor application usage with AppLocker: **https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/monitor-application-usage-with-applocker**.

Exploring Windows Defender Application Control

With its introduction in Windows 10, **Windows Defender Application Control** (**WDAC**) allows organizations to control the applications and drivers that are used in their environment. WDAC is implemented as part of the operating system and was also known under the name **Device Guard**.

It is recommended to use WDAC in combination with **virtualization-based security** (**VBS**). When used with VBS, WDAC's security is enforced by hypervisor isolation, which makes it even harder for an adversary to circumvent your configured application control restrictions. While VBS is technically not required for WDAC, it can significantly enhance your overall system security and should always be enabled if possible.

In comparison to AppLocker rules, WDAC rules are deployed to the whole machine and affect every user logging on to this machine. But WDAC also offers more features and is considered more secure than AppLocker. Its principle is to trust nothing before its trust has been earned.

Applications that are installed from the Microsoft AppStore are, for example, considered trustworthy, as every app that makes it into the store undergoes a strict review process. Default Windows applications are also considered trustworthy and do not need to be separately allowlisted. Other applications can also earn trust via Microsoft Intelligence Security Graph.

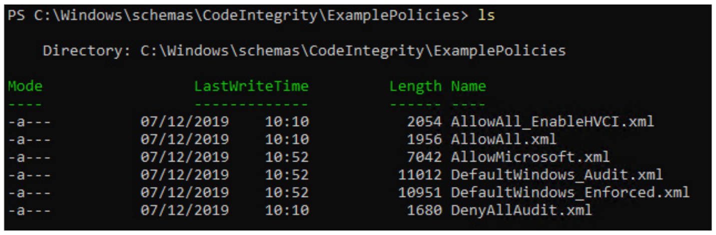
Whether an application is allowed to be executed on a system or not is ensured by so-called **code integrity policies**.

Creating code integrity policies

Code integrity ensures that only trusted system files and drivers are loaded into memory during system boot and runtime. It verifies the digital signatures of files before allowing them to run, and it prevents unsigned or improperly signed files from loading.

The policy with which you configure custom WDAC rules is called a **code integrity policy** (**CI policy**). Similar to other application control mechanisms, it is useful to first deploy your policies in audit mode and monitor for unexpected behaviors before turning on enforcement mode.

On every Windows system that supports WDAC, you can find some example policies under **C:\Windows\schemas\CodeIntegrity\ExamplePolicies**, as shown in the following screenshot:



```
PS C:\Windows\schemas\CodeIntegrity\ExamplePolicies> ls

    Directory: C:\Windows\schemas\CodeIntegrity\ExamplePolicies

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          07/12/2019     10:10           2054 AllowAll_EnableHVCI.xml
-a---          07/12/2019     10:10           1956 AllowAll.xml
-a---          07/12/2019     10:52           7042 AllowMicrosoft.xml
-a---          07/12/2019     10:52          11012 DefaultWindows_Audit.xml
-a---          07/12/2019     10:52          10951 DefaultWindows_Enforced.xml
-a---          07/12/2019     10:10           1680 DenyAllAudit.xml
```

Figure 11.25 – Built-in example code integrity policies

If you create custom policies, it makes sense to start from an existing example policy and then modify it accordingly to build your very own custom policy. The following list will help you determine which **example policy** would be the best base to add your custom rules:

- **AllowAll.xml**: This can be a good base if you are planning to prohibit unwanted applications – you just need to add all deny rules. Please keep in mind that the best option to protect your systems against unauthorized access is to control all applications and only allow the selected ones.
- **AllowAll_EnableHVCI.xml**: By applying this policy, you can enable **memory integrity/hypervisor-protected code integrity** to safeguard against memory attacks. Please refer to the following documentation to learn more about this topic: https://support.microsoft.com/en-us/windows/core-isolation-e30ed737-17d8-42f3-a2a9-87521df09b78.
- **AllowMicrosoft.xml**: This allows Windows, third-party hardware and software kernel drivers, and Windows Store apps, as well as apps that were signed by the Microsoft product root certificate.
- **DefaultWindows_Audit.xml**: Audit mode allows Windows, third-party hardware and software kernel drivers, and Windows Store apps.

- **DefaultWindows_Enforced.xml**: Enforced mode allows Windows, third-party hardware and software kernel drivers, and Windows Store apps but blocks everything else that is not configured.
- **DenyAllAudit.xml**: This policy was created to track all binaries on critical systems – it audits what was to happen if everything was blocked. If enabled, this policy can cause long boot times on Windows Server 2019 operating systems.

In most use cases, the **DefaultWindows_Audit.xml** and **DefaultWindows_Enforced.xml** policies are the best options to create a custom policy and extend them with custom rules as needed.

There's also a list of Microsoft recommended block rules that you should follow: https://learn.**microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/microsoft-recommended-block-rules**.

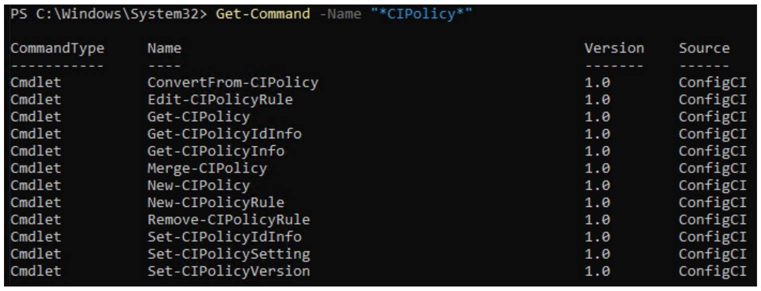
The recommendations in this list can also help you mitigate downgrade attacks. This is an attack in which an attacker uses the older PowerShell v2 to bypass the security features and logging mechanisms of newer versions. We explored this attack in ***Chapter 4***, *Detection – Auditing and Monitoring*.

Although many items on this list may be permitted by default in common policies, it is important to carefully consider what executables and binaries are explicitly needed in your scenario and block all unnecessary ones.

On devices that are managed using Configuration Manager, there is an additional example policy under **C:\Windows\CCM\DeviceGuard**. This policy can be used as a base policy to deploy WDAC policies with Configuration Manager.

Once you have selected an example policy that you want to use as your base, you can start modifying a copy of the selected policy. There are many options that you can configure, so you might want to get started by checking out all the available configuration options in the official documentation: **https://learn.microsoft.com/en-us/windows/security/application-security/application-control/windows-defender-application-control/design/select-types-of-rules-to-create**.

You can either edit an example policy XML file or automate the process of creating code integrity policies using PowerShell. The following screenshot shows which cmdlets are available to operate code integrity policies:



```
PS C:\Windows\System32> Get-Command -Name "*CIPolicy*"

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Cmdlet          ConvertFrom-CIPolicy                               1.0        ConfigCI
Cmdlet          Edit-CIPolicyRule                                  1.0        ConfigCI
Cmdlet          Get-CIPolicy                                       1.0        ConfigCI
Cmdlet          Get-CIPolicyIdInfo                                 1.0        ConfigCI
Cmdlet          Get-CIPolicyInfo                                   1.0        ConfigCI
Cmdlet          Merge-CIPolicy                                     1.0        ConfigCI
Cmdlet          New-CIPolicy                                       1.0        ConfigCI
Cmdlet          New-CIPolicyRule                                   1.0        ConfigCI
Cmdlet          Remove-CIPolicyRule                                1.0        ConfigCI
Cmdlet          Set-CIPolicyIdInfo                                 1.0        ConfigCI
Cmdlet          Set-CIPolicySetting                                1.0        ConfigCI
Cmdlet          Set-CIPolicyVersion                                1.0        ConfigCI
```

Figure 11.26 – Code integrity policy-related cmdlets

One possibility is, for example, the WDAC Policy Wizard, which utilizes the WDAC CI cmdlets that we will look into in the following sections and acts as a wrapper to create CI policies with the help of a GUI. You can download this helpful tool from the official website: https://webapp-wdac-wizard.azurewebsites.net/.

It is also possible to create a custom XML policy using the **New-CIPolicy** cmdlet: one option is to scan a reference system and create a reference XML policy.

Scanning a reference system to create an XML CI policy

The following example shows how to scan the System32 path and the Program Files folder, and subsequently merge both policies into one.

First, let's scan the Windows System32 path:

```
> New-CIPolicy -FilePath "C:\AppControlPolicies\Windows.xml" -Level Publisher -UserPEs -ScanPath "C:\Win
```

While the **-ScanPath** parameter indicates the path that should be scanned by **New-CIPolicy**, the **-UserPEs** parameter indicates that user-mode files will be scanned as well. Only use the **-UserPEs** and **-ScanPath** parameters if you are not providing driver files or rules but want to scan a reference system or path instead.

Using the **-FilePath** parameter, you can specify the output folder where your newly created CI policy should be saved. In this case, we have saved it to **C:\AppControlPolicies\Windows.xml**.

There is also the **-Level** parameter, which indicates the level of the CI pol-icy. Using it, you can specify what kind of files are allowed to run. In this case, the policy is set to the **Publisher** level, which means that all the code must be signed by a trusted publisher so that it can run.

The following levels can also be used:

- **None**: Disables code integrity enforcement. No rules are enforced. This level makes no sense if you want to configure a robust CI policy.
- **Hash**: Allows an application to run only if its hash matches a specified value.
- **FileName**: Allows an application to run only if it is located in a spe-cific file path. This level might sound tempting at first, but it opens up

more risks. If an adversary were to access files on the system, they could easily replace existing files with malicious files. It's best not to use this option.

- **SignedVersion**: Allows an application to run only if it has a specific signed version.
- **Publisher**: Allows an application to run only if it is signed by a specified publisher.
- **FilePublisher**: Allows an application to run only if it is signed by a specified publisher and is located in a specific file path.
- **LeafCertificate**: Allows an application to run only if it is signed by a specified leaf certificate.
- **PcaCertificate**: Allows an application to run only if it is signed by a specified PCA certificate.
- **RootCertificate**: Allows an application to run only if it is signed by a specified root certificate.
- **WHQL**: Allows only signed drivers that are **Windows Hardware Quality Labs** (**WHQL**) certified to be loaded.
- **WHQLPublisher**: Allows only signed drivers that are WHQL certified and signed by a specific publisher to be loaded.
- **WHQLFilePublisher**: Allows only signed drivers that are WHQL certified, signed by a specific publisher, and located in a specific file path to be loaded.

Next, let's scan the **Program Files** folder to create a policy from the specified reference system:

```
> New-CIPolicy -FilePath "C:\AppControlPolicies\ProgramFiles.xml" -Level Publisher -UserPEs -ScanPath "C
```

Again, we have included our user-mode files in the scan and want to ensure that all the files included in our policy are signed by a specified publisher. We must define that the newly created policy will be saved to **C:\AppControlPolicies\ProgramFiles.xml**. To avoid script files from being included in this reference policy, we must specify the **-NoScript** parameter.

Using the **-Fallback** parameter, you can specify a fallback order; in this case, if there is no match at the **FilePublisher** level, the policy engine will fall back to the **SignedVersion**, **FilePublisher**, and **Hash** levels – exactly in this order.

Last, but not least, we need to merge the policies into one. To do so, we can use the **Merge-CIPolicy** cmdlet:

```
> Merge-CIPolicy -PolicyPaths "C:\AppControlPolicies\Windows.xml", "C:\AppControlPolicies\ProgramFiles.x
```

Using the **-PolicyPaths** parameter, we can specify which policies should be merged, while with **-OutputFilePath**, we can define where the merged policy will be saved to. In this example, we'll save the final policy under **C:\AppControlPolicies\AppControlPolicy.xml**.

The policy is created in audit mode so that it can't block and only audit the use of applications. This is especially useful for testing and evaluating what applications should be blocked.

Once you are ready to apply a block policy to your systems, you can remove the audit-only configuration from your policy using the following command:

```
> Set-RuleOption -FilePath "C:\AppControlPolicies\AppControlPolicy.xml" -Option 3 -Delete
```

To deploy your newly generated policy, you will need to convert it into binary format.

Converting the XML file into a binary CI policy

Once you have obtained your CI policy XML configuration file, you will need to convert it into binary format to deploy it. This can be done using the **ConvertFrom-CIPolicy** cmdlet:

```
> ConvertFrom-CIPolicy -XmlFilePath "C:\AppControlPolicies\AppControlPolicy.xml" -BinaryFilePath "C:\Win
```

Here, the **AppControlPolicy.xml** CI policy, which we generated earlier, will be compiled into the **AppControlPolicy.bin** binary file and saved under **C:\Windows\System32\CodeIntegrity\AppControlPolicy.bin**.

If a binary CI policy is saved under **C:\Windows\System32\CodeIntegrity\**, it will be enabled immediately after the affected system is restarted. Once the policy is removed again and the system is restarted, all changes introduced by the CI policy are reverted.

Of course, you can also save the converted CI policy under another path of your choice if you plan to deploy WDAC using Intune, MEM, GPO, or another deployment mechanism that requires a binary configuration file.

There are also other ways to create a CI policy XML file – for example, from audited events.

Using audited events from the event log as a reference

Another way to create a WDAC policy is by running WDAC in audit mode and using the audit log to create the policy. Similar to AppLocker, if WDAC is running in audit mode, any application that would be blocked if the current WDAC configuration was enabled is logged to the audit log.

Depending on the application type, these events can be found in one of the following event logs:

- **Binary-related events**: **Applications and Services Logs** | **Microsoft** |**Windows** | **CodeIntegrity** | **Operational**

- **MSI and script-related events**: **Applications and Services Logs** | **Microsoft** | **Windows** | **AppLocker** | **MSI and Script**

All events logged to these event logs can now be leveraged to either create a completely new CI policy or to merge audited configurations into an existing policy:

```
> New-CIPolicy -FilePath "C:\AppControlPolicies\AuditEvents.xml" -Audit -Level FilePublisher -Fallback S
```

This command creates a new CI policy under the **C:\AppControlPolicies\AuditEvents.xml** path. The **-Audit** parameter specifies that the actual audit events from the event log should be used to create the policy.

The **-MultiplePolicyFormat** parameter enables us to use multiple policies at the same time since the policy will be stored in a multiple-policy format, as introduced in Windows 10.

Now, you can review and edit the newly created policy before merging it with other existing policies and/or converting it into binary format for further use.

Creating a CI policy using the New-CIPolicyRule cmdlet

If you want to define what applications should appear in your CI policy more granularly, the **New-CIPolicyRule** cmdlet can help you out:

```
> $Rules = New-CIPolicyRule -FilePathRule "C:\Program Files\Notepad++\*"
```

```
> $Rules += New-CIPolicyRule -FilePathRule "C:\Program Files\PowerShell\7\*"
```

```
> New-CIPolicy -Rules $Rules -FilePath "C:\AppControlPolicies\GranularAppControlPolicy.xml" -UserPEs
```

The preceding code would create one CI policy rule for the *Notepad++* folder and its subfolders, as well as one for the *PowerShell 7* path, and saves both rules in the **$Rules** variable.

Then, both rules can be used to create a new CI policy that is saved under the **C:\AppControlPolicies\GranularAppControlPolicy.xml** path.

Later, you can either combine it with other policies using **Merge-CIPolicy** or convert it into binary format with the help of **ConvertFrom-CIPolicy** so that you can use it for other purposes.

You can use the ConfigCI PowerShell module to explore other ways of working with code integrity: https://learn.microsoft.com/en-us/powershell/module/configci.

Although it is not technically required, virtualization-based security features such as Secure Boot should be enabled so that code integrity functions properly. Secure Boot ensures that the system only boots to a trusted state, and that all boot files are signed with trusted signatures. This prevents the boot process from being tampered with and ensures the integrity of the operating system and its drivers.

Virtualization-based security (VBS)

VBS uses virtualization as a base to isolate areas in memory from the *normal* operating system. By doing this, the isolated area can be protected in a better way by encrypting the available memory and the communication to and from this memory area.

Through this isolation, those memory areas can be better protected against vulnerabilities that are active in the operating system.

One example of this is protecting credentials in the **local security authority** (**LSA**), which makes it harder to extract and steal credentials from the operating system.

Another example is **hypervisor-protected code integrity** (**HVCI**), which uses VBS for code integrity.

Hypervisor-protected code integrity (HVCI)

HVCI, also called **memory integrity**, is the key component of VBS. HVCI leverages VBS technology to protect against kernel-mode attacks by ensuring the integrity of the kernel and critical system components. It does so by allowing only trusted and authorized code to run in kernel mode.

If HVCI is active, the CI functionality is forwarded to a secure virtual environment on the same machine, in which the WDAC functionality itself is executed to ensure integrity. As mentioned previously, HVCI uses VBS technology to protect against kernel-mode attacks. It enforces the integrity of the kernel and critical system components by verifying that only known and trusted code can run in kernel mode. But technically, VBS is not required for WDAC.

HVCI utilizes hardware features such as virtualization extensions in modern CPUs and the **Trusted Platform Module** (**TPM**) to create a secure execution environment. The TPM is used to store a hash of the system's boot firmware, UEFI, and operating system binaries. During system boot, the TPM measures these components and provides the measurements to the HVCI system. HVCI uses these measurements to verify that only known and trusted components are loaded into memory, thus preventing unauthorized code from running in kernel mode.

If you want to enable HVCI options for a CI policy, you can use the **Set-HVCIOptions** cmdlet:

```
> Set-HVCIOptions -Enabled -FilePath "C:\AppControlPolicies\GranularAppControlPolicy.xml"
```

You can take this even further by using the **-Strict** parameter:

```
> Set-HVCIOptions -Strict -FilePath "C:\AppControlPolicies\GranularAppControlPolicy.xml"
```

If the **-Strict** option is used, this means that only Microsoft and WHQL-signed drivers will be allowed to load after this policy is applied.

To remove all HVCI settings from a CI policy, you can specify the **-None** parameter:

```
> Set-HVCIOptions -None -FilePath "C:\AppControlPolicies\GranularAppControlPolicy.xml"
```

Another helpful VBS feature is Secure Boot, which helps you significantly enhance the security of your Windows systems.

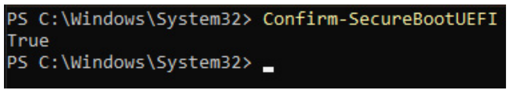
Enabling Secure Boot

Secure Boot ensures that the system is booted into a trusted state. This means that all files that are used to boot the system need to be signed with signatures that are trusted by the organization. By doing this, the system will not be booted if those files have been tampered with. The device needs to have a TPM chip to support Secure Boot.

To verify if Secure Boot is enabled on your computer, you can utilize the **Confirm-SecureBootUEFI** cmdlet:

```
> Confirm-SecureBootUEFI
```

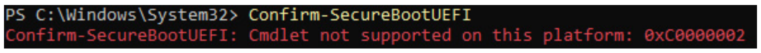
If Secure Boot is enabled, the cmdlet will return **True**, as shown in the following screenshot; if not, **False** will be returned:



```
PS C:\Windows\System32> Confirm-SecureBootUEFI
True
PS C:\Windows\System32> _
```

Figure 11.27 – Secure Boot is enabled

If the hardware of your PC does not support Secure Boot, you will receive an error message stating **Cmdlet not supported on this platform.**:



```
PS C:\Windows\System32> Confirm-SecureBootUEFI
Confirm-SecureBootUEFI: Cmdlet not supported on this platform: 0xC0000002
```

Figure 11.28 – The hardware does not support Secure Boot

Have a look at the following links if you want to learn more about Secure Boot:

- *Secure Boot*: **https://learn.microsoft.com/en-us/powershell/module/secureboot**
- *Secure Boot Landing*: https://learn.microsoft.com/en-us/windows-hardware/manufacture/desktop/secure-boot-landing

Adversaries often use malicious drivers and manipulated system files. Secure Boot, when combined with code integrity, ensures that the booted operating system, as well as its used drivers, can be trusted.

Deploying WDAC

There are different ways to deploy WDAC: MDM or Intune, Configuration Manager, GPO, and PowerShell.

As describing every deployment method in detail would exceed the capacity of this book, please refer to the official deployment guide, where you can find detailed instructions for every deployment method: https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/windows-defender-application-control-deployment-guide.

In the following sections, we will explore the pros and cons of each different deployment method.

GPO

Group Policy is not the preferred method to configure WDAC; it only supports single-policy format **CI policies** with a **.bin**, **.p7b**, or **.p7** file type. This format was used for devices before Windows 10 version 1903. As a best practice, use a deployment mechanism other than GPO.

However, if you want to use this deployment method anyway, you can find the WDAC GPO setting under **Computer Configuration** | **Administrative Templates** | **System** | **Device Guard** | **Deploy Windows Defender Application Control**. Using this, you can deploy a CI policy.

The binary CI policy that you want to deploy needs to be located either on a file share or copied to the local system of each machine that you want to restrict.

Detailed documentation on how to deploy WDAC using GPO can be found here: **https://learn.microsoft.com/en-us/windows/security/application-security/application-control/windows-defender-application-control/deployment/deploy-wdac-policies-using-group-policy**.

Intune

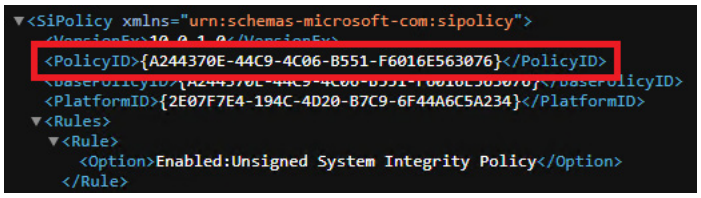
You can use an MDM solution to configure WDAC, such as Intune. Using Intune, application control comes with some built-in policies that you can configure so that your clients can only run Windows components, third-party hardware and software kernel drivers, apps from the Microsoft store, and applications with a good reputation that are trusted by Microsoft Intelligence Security Graph (optional).

Of course, it is also possible to create custom WDAC policies using OMA-URI, which can be done similarly to configuring AppLocker policies using Intune.

In every XML CI policy file, you can find a policy ID. Copy this ID and replace **{PolicyID}** in the following string to get the OMA-URI for your custom policy:

```
./Vendor/MSFT/ApplicationControl/Policies/{PolicyID}/Policy
```

Please note that you also need to replace the curly brackets. The following screenshot shows where you can find **PolicyID**:



```
▼<SiPolicy xmlns="urn:schemas-microsoft-com:sipolicy">
   <VersionEx>10.0.1.0</VersionEx>
   <PolicyID>{A244370E-44C9-4C06-B551-F6016E563076}</PolicyID>
   <BasePolicyID>{A244370E-44C9-4C06-B551-F6016E563076}</BasePolicyID>
   <PlatformID>{2E07F7E4-194C-4D20-B7C9-6F44A6C5A234}</PlatformID>
 ▼<Rules>
   ▼<Rule>
       <Option>Enabled:Unsigned System Integrity Policy</Option>
     </Rule>
```

Figure 11.29 – You can find the policy ID in the XML CI policy file

Using this **PolicyID**, the corresponding OMA-URI would be as follows:

```
./Vendor/MSFT/ApplicationControl/Policies/A244370E-44C9-4C06-B551-F6016E563076/Policy
```

You can learn more about how to use Intune for deploying WDAC at **https://learn.microsoft.com/en-us/windows/security/application-security/application-control/windows-defender-application-control/deployment/deploy-wdac-policies-using-intune**.

Microsoft Configuration Manager

When using Configuration Manager, it becomes a trustworthy source itself. This means that every application and piece of software that was installed over Configuration Manager becomes trustworthy and is allowed to run. This option needs to be configured through a built-in policy first.

Similar to deploying with Intune, Configuration Manager also provides some more built-in policies so that you can configure your clients to only run Windows components and apps from the Microsoft Store. It is also optional to trust apps with a good reputation, verified by the **Intune Service Gateway (ISG)**. Configuration Manager comes with another optional built-in policy: it is possible to allow apps and other executables that were already installed in a defined folder.

You can learn more about WDAC can be deployed using Configuration Manager at **https://learn.microsoft.com/en-us/windows/security/application-security/application-control/windows-defender-application-control/deployment/deploy-wdac-policies-with-memcm**.

PowerShell

Depending on the operating system, there are different ways to deploy WDAC using PowerShell since not all capabilities are available for every operating system version. The **WDAC policy refresh tool** also needs to be downloaded and deployed to every managed endpoint: **https://www.microsoft.com/en-us/download/details.aspx?id=102925**.

For this method, you will also need the policy's binary to copy it to each managed endpoint. However, compared to GPO, you can deploy multiple WDAC policies. To deploy signed policies, you will also need to copy the binary policy file to the device's EFI partition. Signed policies provide an additional layer of security by ensuring that only policies signed by trusted entities are applied to the endpoint. This step will be done automatically if Intune or the CSP is used for deployment.

Matt Graeber's **WDACTools** is also a valuable resource for streamlining your deployment process. These tools were specifically designed to simplify the process of building, configuring, deploying, and auditing WDAC policies. You can download them from Matt's GitHub repository: **https://github.com/mattifestation/WDACTools**.

For detailed information on how to deploy WDAC using PowerShell, please refer to **https://learn.microsoft.com/en-us/windows/security/application-security/application-control/windows-defender-application-control/deployment/deploy-wdac-policies-with-script**.

How does PowerShell change when application control is enforced?

When application control is enforced, PowerShell acts as a safeguard to prevent the misuse of its features by potential adversaries. By proactively implementing application control measures, PowerShell ensures that its powerful scripting language cannot be easily abused by attackers to bypass imposed restrictions.

PowerShell can be restricted in several ways, including disabling the ability to run PowerShell scripts or only allowing signed PowerShell scripts to run.

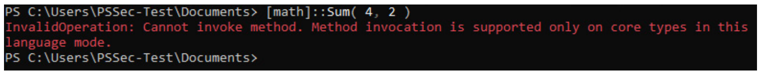
In ***Chapter 5***, *PowerShell Is Powerful – System and API Access*, we discussed how it is possible to use PowerShell to run arbitrary **.NET** code or even execute compiled code if the system is not restricted. This can make it very difficult to protect against malicious code. With application control enforced, it's possible to eliminate unconstrained code execution methods such as **Add-Type**, arbitrary .NET scripting, and other options that are typically used to bypass security mechanisms.

PowerShell includes a built-in **Constrained Language mode**, which we explored in ***Chapter 10***, *Language Modes and Just Enough Administration (JEA)*. Constrained Language mode limits PowerShell and restricts the user from executing risky language elements, such as accessing arbitrary APIs.

This means that certain *dangerous* language elements such as **Add-Type**, **COM objects**, and some .NET types that can be utilized to execute arbitrary code cannot be used. If enforced, Constrained Language mode can limit the attacker's ability to execute arbitrary code and modify system configurations. In Constrained Language mode, the PowerShell environment retains only the core basic features of a traditional less powerful interactive shell, similar to CMD, Windows Explorer, or Bash.

One effective approach to ensure that PowerShell code is trusted is to enforce the use of **signed scripts**. With application control in place, if a script is trusted and allowed to run in **Full Language mode**, it is executed accordingly. But if it is not trusted, a script will always run in Constrained Language mode, which means that the script will fail if it attempts to call arbitrary APIs and other risky language elements.

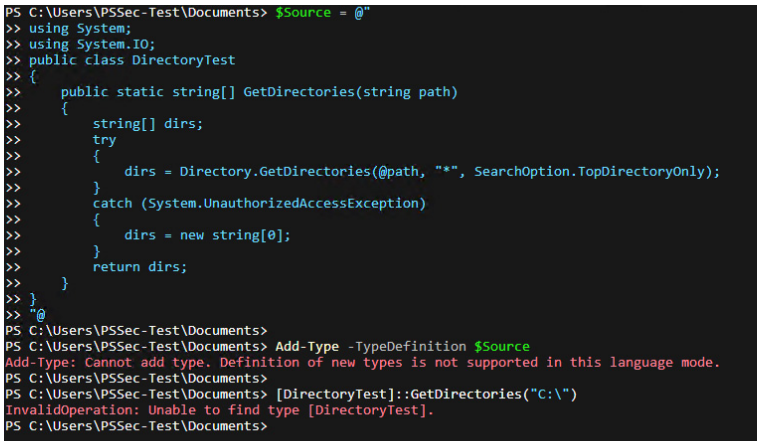
When application control is enforced, and therefore PowerShell were to run in Constrained Language mode, if you were to try to call methods directly from .NET, they would fail, as shown in the following screenshot:



```
PS C:\Users\PSSec-Test\Documents> [math]::Sum( 4, 2 )
InvalidOperation: Cannot invoke method. Method invocation is supported only on core types in this
language mode.
PS C:\Users\PSSec-Test\Documents>
```

Figure 11.30 – .NET types cannot be accessed with application control enabled

Using **Add-Type** to add and access your C types from PowerShell would also not work – you would get the following error message:



```
PS C:\Users\PSSec-Test\Documents> $Source = @"
>> using System;
>> using System.IO;
>> public class DirectoryTest
>> {
>>     public static string[] GetDirectories(string path)
>>     {
>>         string[] dirs;
>>         try
>>         {
>>             dirs = Directory.GetDirectories(@path, "*", SearchOption.TopDirectoryOnly);
>>         }
>>         catch (System.UnauthorizedAccessException)
>>         {
>>             dirs = new string[0];
>>         }
>>         return dirs;
>>     }
>> }
>> "@
PS C:\Users\PSSec-Test\Documents>
PS C:\Users\PSSec-Test\Documents> Add-Type -TypeDefinition $Source
Add-Type: Cannot add type. Definition of new types is not supported in this language mode.
PS C:\Users\PSSec-Test\Documents>
PS C:\Users\PSSec-Test\Documents> [DirectoryTest]::GetDirectories("C:\")
InvalidOperation: Unable to find type [DirectoryTest].
PS C:\Users\PSSec-Test\Documents>
```

Figure 11.31 – Add-Type fails when application control is enforced

These are not the only commands that would fail, but they should demonstrate how the PowerShell experience is different with application control enabled.

If you allow signed Windows files with your application control policy, this means that PowerShell modules that come with your Windows installation will also be allowed to run in Full Language mode. However, custom-created modules would run in Constrained language mode, unless they have been configured to be trusted in your application control setup. This effectively reduces the attack surface of the system.

As mentioned earlier in this chapter, at the time of writing, PowerShell and the WSH family are the only dynamic runtimes that can be restricted using application control, while others still allow unrestricted code execution. Therefore, PowerShell is a huge advantage when locking down your environment with application control policies.

In summary, enforcing application control mechanisms such as WDAC and AppLocker can have a significant impact on improving PowerShell security. It's possible to limit the ability of PowerShell scripts to execute arbitrary code or modify system configurations by enforcing constraints such as Constrained Language mode. By implementing these measures, it's possible to reduce the attack surface of the system significantly and make it more difficult for attackers to execute malicious code.

Summary

In this chapter, you learned how to configure your existing PowerShell scripts as trustworthy and how to allowlist them, but not just PowerShell scripts. At this point, you should have a good understanding of how you can implement a proper application control solution for all the applications in your environment.

First, you explored how to sign your code and how to create a self-signed script that you can use for testing purposes. With this knowledge, you can easily transfer to your enterprise scenario, in which you might already have corporate-signed or public-signed certificates in use.

Next, we dove into application control and learned what built-in application control solutions exist: SRP, AppLocker, and WDAC. You should now also be familiar with how to plan for allowlisting applications in your environment.

Then, we explored AppLocker and WDAC and learned how to audit AppLocker and WDAC. We also investigated how to configure AppLocker to avoid a possible PowerShell downgrade attack.

Last but not least, we learned that whenever possible, WDAC is the most secure option, followed by AppLocker. However, both can be combined in the same environment, depending on your operating systems and use cases.

However, only restricting scripts and applications is not enough for a secure and hardened environment. In the next chapter, we'll explore how the Windows **Antimalware Scan Interface** (**AMSI**) can protect you from malicious code that is run directly in the console or in memory.

Further reading

If you want to explore some of the topics that were mentioned in this chapter, take a look at the following resources:

Certificate operations:

- New-SelfSignedCertificate: **https://docs.microsoft.com/en-us/powershell/module/pki/new-selfsignedcertificate**

- Set-AuthenticodeSignature: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-authenticodesignature**
- Get-AuthenticodeSignature: **https://docs.microsoft.com/en-us/powershell/wwmodule/microsoft.powershell.security/get-authenticodesignature**

CI/CD:

- CI/CD: The what, why, and how: **https://resources.github.com/ci-cd/**
- About continuous integration: **https://docs.github.com/en/actions/automating-builds-and-tests/about-continuous-integration**

Application control:

- Application Control for Windows: **https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/windows-defender-application-control**
- Authorize reputable apps with the Intelligent Security Graph (ISG): **https://learn.microsoft.com/en-us/windows/security/application-security/application-control/windows-defender-application-control/design/use-wdac-with-intelligent-security-graph**
- Enable virtualization-based protection of code integrity: **https://learn.microsoft.com/en-us/windows/security/hardware-security/enable-virtualization-based-protection-of-code-integrity**
- ConfigCI module reference (ConfigCI): **https://docs.microsoft.com/en-us/powershell/module/configci**
- Understand Windows Defender Application Control (WDAC) policy rules and file rules: **https://learn.microsoft.com/en-us/windows/security/application-security/application-control/windows-defender-application-control/design/select-types-of-rules-to-create**
- Understanding WDAC Policy Settings: **https://learn.microsoft.com/en-us/windows/security/application-security/application-control/windows-defender-application-control/design/understanding-wdac-policy-settings**
- Use multiple Windows Defender Application Control Policies: **https://learn.microsoft.com/en-us/windows/security/application-security/application-control/windows-defender-application-control/design/deploy-multiple-wdac-policies**
- Use signed policies to protect Windows Defender Application Control against tampering: **https://learn.microsoft.com/en-us/windows/security/application-security/application-control/windows-defender-application-control/deployment/use-signed-policies-to-protect-wdac-against-tampering**
- Windows Defender Application Control management with Configuration Manager: **https://learn.microsoft.com/en-us/mem/configmgr/protect/deploy-use/use-device-guard-with-configuration-manager**
- Windows Defender Application Control Wizard: **https://learn.microsoft.com/en-us/windows/security/application-**

security/application-control/windows-defender-application-control/design/wdac-wizard

AppLocker:

- AppLocker Operations Guide: **https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-R2-and-2008/ee791916(v=ws.10)**
- Enable the DLL rule collection: **https://learn.microsoft.com/en-us/windows/security/application-security/application-control/windows-defender-application-control/applocker/enable-the-dll-rule-collection**

You can also find all the links mentioned in this chapter in the GitHub repository for ***Chapter 11*** – no need to manually type in every link: https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter11/Links.md.