Application whitelisting with “AaronLocker”

Automated AppLocker policy generation and maintenance

Aaron Margosis

Microsoft Cybersecurity Services

May 2019

# Overview

AaronLocker is designed to make the creation and maintenance of robust, strict, AppLocker-based whitelisting rules as easy and practical as possible. The entire solution involves a small number of PowerShell scripts. You can easily customize rules for your specific requirements with simple text-file edits. AaronLocker includes scripts that document AppLocker policies and capture event data into Excel workbooks that facilitate analysis and policy maintenance.

AaronLocker is designed to restrict program and script execution by non-administrative users. Note that AaronLocker does not try to stop administrative users from running anything they want – and AppLocker cannot meaningfully restrict administrative actions anyway. A determined user with administrative rights can easily bypass AppLocker rules.

AaronLocker’s strategy can be summed up as: if a non-admin could have put a program or script onto the computer – i.e., it is in a user-writable directory – don’t allow it to execute unless it has already been specifically allowed by an administrator. This will stop execution if a user is tricked into downloading malware, if an exploitable vulnerability in a program the user is running tries to put malware on the computer, or if a user intentionally tries to download and run unauthorized programs.

AaronLocker works on all supported versions of Windows that can provide AppLocker.

Part I of this document is a high-level description of whitelisting concepts, AppLocker, and the AaronLocker approach. Part II is the “operations guide” that digs into the details of implementing AaronLocker for your environment.

A personal note: the name “AaronLocker” was Chris (@appcompatguy) Jackson’s idea – not mine – and I resisted it for a long time. I finally gave in because I couldn’t come up with a better name.

Table of Contents

[Overview i](#_Toc8513457)

[Part I – AaronLocker philosophy and basics 1](#_Toc8513458)

[Whitelisting 1](#_Toc8513459)

[Whitelisting strategies 1](#_Toc8513460)

[Intro to AppLocker 2](#_Toc8513461)

[Rule collections 2](#_Toc8513462)

[Rule types and when to use them 3](#_Toc8513463)

[Path rules 4](#_Toc8513464)

[Publisher rules 4](#_Toc8513465)

[Hash rules 5](#_Toc8513466)

[“Administrators” and “Everyone” 5](#_Toc8513467)

[AppLocker GUI, default/generated rules and their limitations 6](#_Toc8513468)

[Intro to AaronLocker 6](#_Toc8513469)

[Part II – AaronLocker Operations Guide 9](#_Toc8513470)

[Getting started – building the first rule sets 9](#_Toc8513471)

[Setting up 9](#_Toc8513472)

[AaronLocker contents 9](#_Toc8513473)

[Defining and tuning “user-writable” 11](#_Toc8513474)

[Initial customizations 12](#_Toc8513475)

[Safe paths 12](#_Toc8513476)

[Unsafe paths 13](#_Toc8513477)

[Trusted Signers 14](#_Toc8513478)

[Customizing ExeFilesToBlacklist 15](#_Toc8513479)

[Merge additional policy fragments 15](#_Toc8513480)

[Getting the data for first customizations 15](#_Toc8513481)

[Build AaronLocker policies for pilot testing 18](#_Toc8513482)

[Next steps 18](#_Toc8513483)

[Tip: moving AaronLocker policy management to another computer 18](#_Toc8513484)

[Applying AaronLocker through local policy 19](#_Toc8513485)

[Deploying AaronLocker through Active Directory and Group Policy Objects 19](#_Toc8513486)

[Security groups 19](#_Toc8513487)

[Group Policies 20](#_Toc8513488)

[Configuring GPO settings 20](#_Toc8513489)

[Configuring GPO Security Filtering 20](#_Toc8513490)

[Configuring the Windows Event Collector 27](#_Toc8513491)

[Evaluating results and tuning the rule set 28](#_Toc8513492)

[Using Get-AppLockerEvents.ps1 29](#_Toc8513493)

[Get-AppLockerEvents.ps1 to Excel 29](#_Toc8513494)

[Get-AppLockerEvents.ps1’s CSV output 29](#_Toc8513495)

[Get-AppLockerEvents.ps1’s PSCustomObject output 29](#_Toc8513496)

[Event data and synthesized data 30](#_Toc8513497)

[Event types and sources 33](#_Toc8513498)

[“Noise” events and “filtered machines” 33](#_Toc8513499)

[Using Save-WEFEvents.ps1 33](#_Toc8513500)

[Using Generate-EventWorkbook.ps1 34](#_Toc8513501)

[Making adjustments 35](#_Toc8513502)

[Altering the program 35](#_Toc8513503)

[Customizing the rules 36](#_Toc8513504)

[Whitelisting beyond AppLocker 37](#_Toc8513505)

[Microsoft Office macros 37](#_Toc8513506)

[Macros embedded in a Word, Excel, or PowerPoint document 37](#_Toc8513507)

[User-defined macros stored in the user’s profile 39](#_Toc8513508)

[The VBA editor’s “Immediate window” 39](#_Toc8513509)

[Oracle Java and other programming platforms 39](#_Toc8513510)

[Unusual EXE/DLL combinations 40](#_Toc8513511)

[Appendix A: Main script reference 41](#_Toc8513512)

[Create-Policies.ps1 41](#_Toc8513513)

[Scan-Directories.ps1 44](#_Toc8513514)

[ExportPolicy-ToExcel.ps1 47](#_Toc8513515)

[Compare-Policies.ps1 49](#_Toc8513516)

[Get-AppLockerEvents.ps1 51](#_Toc8513517)

[Save-WEFEvents.ps1 58](#_Toc8513518)

[Generate-EventWorkbook.ps1 59](#_Toc8513519)

[Appendix B: CustomizationInputs and MergeRules-\* reference 60](#_Toc8513520)

[GetExeFilesToBlacklist.ps1 60](#_Toc8513521)

[GetSafePathsToAllow.ps1 60](#_Toc8513522)

[HashRuleData.ps1 61](#_Toc8513523)

[KnownAdmins.ps1 61](#_Toc8513524)

[TrustedSigners.ps1 62](#_Toc8513525)

[TrustedSigners-MsvcMfc.ps1 63](#_Toc8513526)

[UnsafePathsToBuildRulesFor.ps1 64](#_Toc8513527)

[The MergeRules-Dynamic and MergeRules-Static directories 65](#_Toc8513528)

[MergeRules-Dynamic 65](#_Toc8513529)

[MergeRules-Static 66](#_Toc8513530)

[Appendix C: Support scripts reference 67](#_Toc8513531)

[Set-OutputEncodingToUnicode.ps1 67](#_Toc8513532)

[DownloadAccesschk.ps1 67](#_Toc8513533)

[Get-AaronLockerTimestamp.ps1 67](#_Toc8513534)

[Enum-WritableDirs.ps1 68](#_Toc8513535)

[BuildRulesForFilesInWritableDirectories.ps1 70](#_Toc8513536)

[ExportPolicy-ToCsv.ps1 72](#_Toc8513537)

[Other files 73](#_Toc8513538)

[Appendix D: PowerShell and whitelisting 74](#_Toc8513539)

[Appendix E: Troubleshooting tips 76](#_Toc8513540)

[PowerShell script errors 76](#_Toc8513541)

[*scriptname*.ps1 cannot be loaded because running scripts is disabled on this system. 76](#_Toc8513542)

[*scriptname*.ps1 cannot be loaded. The file *scriptname*.ps1 is not digitally signed. You cannot run this script on the current system. 76](#_Toc8513543)

[*scriptname*.ps1 cannot be loaded because its operation is blocked by software restriction policies, such as those created by using Group Policy. 76](#_Toc8513544)

# Part I – AaronLocker philosophy and basics

Part I of this document is a high-level description of whitelisting concepts, AppLocker, and the AaronLocker approach.

## Whitelisting

“Whitelisting” defines precisely what a user is allowed to run and disallows everything else. This is contrasted with “blacklisting,” which blocks specific programs while allowing everything else, and traditional anti-malware, which examines each piece of code and dynamically decides whether to allow it to execute.

Application whitelisting is a powerful defense against malware, including ransomware, and has been widely advocated by security experts. Properly configured, whitelisting can prevent users from being tricked into running malicious code and from intentionally downloading and running unauthorized programs, and can severely limit what an exploited vulnerability in a user program can accomplish.

Windows includes strong whitelisting technologies, including AppLocker beginning with Windows 7 and Windows Server 2008 R2, and Windows Defender Application Control[[1]](#footnote-2) (WDAC) beginning in Windows 10 and Windows Server 2016. However, most organizations find it difficult to define, apply, and maintain effective policies that don’t negatively impact productivity. The AaronLocker solution offers an automated process to generate whitelisting rules that are robust, practical, and easily maintainable. It makes it easier to define and maintain rules that allow execution only from safe locations, disallowing the common whitelisting bypasses, while creating tightly-scoped exceptions for programs that must run from such "unsafe" locations.

## Whitelisting strategies

Some of the questions raised in a whitelisting effort are:

1. Are we concerned only about non-admins or do we also need to be able to restrict what administrators and the operating system run?
2. What threats are we most concerned with? How much do we need to worry about a malicious user at the keyboard?
3. How granularly must we define the rules?

For question 1: Restricting administrators is vastly more difficult than restricting non-administrators. Within the Windows universe (at least), administrators have full control over the operating system. Anything the operating system can do an administrator can do as well.[[2]](#footnote-3) Because AppLocker rules are enforced by the operating system, AppLocker can be subverted very easily in a variety of ways by an actor with administrative rights. For these reasons, AaronLocker restricts only non-administrators and does not try to restrict administrators.

For question 2: The low bar with whitelisting is to defend against drive-by download attacks (such as through browser vulnerabilities) and against naïve users getting tricked into downloading and running malicious programs. That by itself is a huge step forward for most organizations. The higher bar is to defend against a malicious (or *extremely* gullible) interactive user at the keyboard willing to type complex command lines into a command processor. (Public kiosk systems should be considered to face this risk.) The internet offers many lists of whitelisting bypasses that first require having acquired this level of control. Those are harder to defend, but in many cases are substantially less likely to be a legitimate threat. AaronLocker tries to incorporate some defenses against these additional threats.

For question 3: Some tightly-scoped whitelisting strategies can be summarized as, “Identify every single individual file that users might need to execute, and create rules covering each one and nothing else.” It can be challenging to create the initial rule set correctly, and far more challenging to maintain the rule set as existing files are updated and new applications are installed. Once the decision has been taken to focus on restricting only non-administrators, an almost equally effective but much more manageable strategy is to assume that a program can be trusted for execution if an administrator put it on the computer in a location that is accessible but read-only to non-administrators, and that if a non-administrator could have put it on the computer (i.e., it is in a user-writable directory) that it should not be trusted unless by an explicit exception. AaronLocker is based on this strategy. The US National Security Agency’s (NSA) Information Assurance Directorate (IAD) published a whitepaper describing such a strategy[[3]](#footnote-4), which is also the foundational strategy for AaronLocker.

## Intro to AppLocker

AppLocker is a whitelisting technology that was first introduced in Windows 7 and in Windows Server 2008 R2. This section gives an overview of AppLocker features that are particularly relevant to the AaronLocker solution.

AppLocker was designed to control execution only within interactive user sessions, and not to apply to services or other processes in “session 0.” Recently, Windows extended the AppLocker policy XML schema to support the “managed installer” feature used by Windows Defender Application Control. The policy XML can now be manually edited so that AppLocker can be engaged in session 0. Still, because as mentioned earlier, AppLocker cannot meaningfully restrict administrator actions, the work to try to control session-0 execution using AppLocker is not worth the effort.

### Rule collections

AppLocker divides rules into separate collections based on file type. Note that while “file type” is often associated with file extension, AppLocker rules are applied based on how the file is used, regardless of extension. For example, while executables normally have a .exe extension, they can have any extension or no extension at all.

AppLocker’s five rule collections are:

* Executable rules – these rules come into play at the start of a new process and determine whether Windows allows that process to run.
* Windows Installer rules – these rules determine whether MsiExec.exe will process a Windows installer file such as a .msi or .msp.
* Script rules – these rules determine whether and how Windows script engines and command processors handle script and batch files. It also determines whether PowerShell consoles run in ConstrainedLanguage or FullLanguage mode.
* Packaged App rules – these rules determine which “modern apps” (a.k.a., “Windows Store apps” and on Windows 10, “Universal Windows Platform (UWP) apps”) are allowed to run. Packaged App rules are not applicable on Windows 7 and Windows Server 2008 R2.
* DLL rules – these rules control which DLLs and other mapped files can be loaded into a running process.

You can turn each collection on or off. When you turn one on, you can specify whether it enforces rules or is “audit only.” When someone tries to run a file that isn’t allowed by a matching rule, “enforce” mode prevents it from running and writes an error to the event log. In audit mode, the file is allowed to execute, and a warning is written to the event log indicating that it would have been blocked in enforce mode.

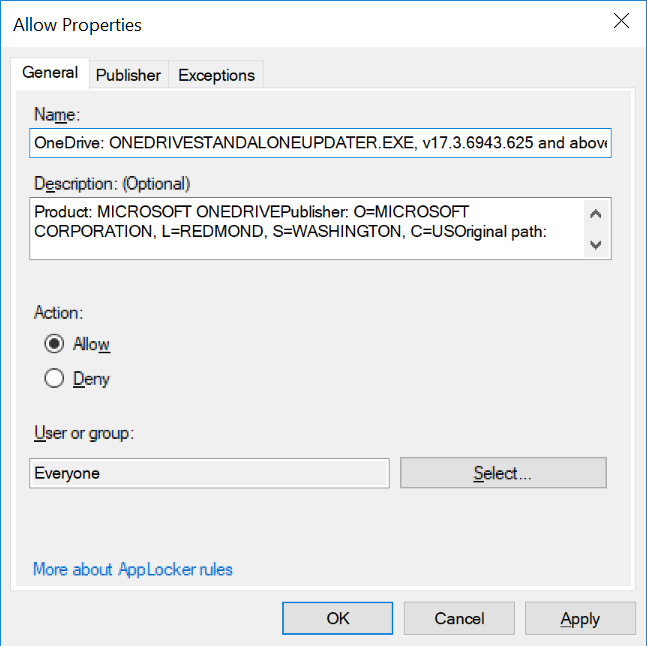
Note: with PowerShell v5.x, when AppLocker reports an error or warning for a PowerShell script, it does not mean that the script was not allowed to execute. It means only that it ran in ConstrainedLanguage mode. See the Appendix about PowerShell and whitelisting for more information.

Note that on Windows 8.1 and Windows 10, the Start screen/menu is a “packaged app” and won’t work correctly if that rule collection is not properly configured. For that reason, and because each modern app runs in its own highly-constrained AppContainer and presents no practical risk to user data or to the system, AaronLocker does not limit packaged-app execution.[[4]](#footnote-5)

The DLL rule collection is not enabled by default. The AppLocker GUI displays dire warnings about performance impact when enabling the DLL rule collection. I recommend ignoring that warning. It might have been applicable when the UI was first designed about a dozen years ago.

### Rule types and when to use them

When file execution is requested and the file’s corresponding rule collection is configured, Windows tries to determine whether the collection contains a matching rule that applies to the file. Each AppLocker rule defines criteria to match against the file. AppLocker defines three criteria types: Path, Publisher, and Hash. Each AppLocker rule also has a name and description, an action (allow or deny), and the user or group to which the rule is applied. Rules can also define exceptions.



If a rule’s criteria matches the file and the user, the rule’s action (“allow” or “deny”) determines whether the file is allowed to execute or is blocked. Whitelisting is built primarily around “allow” rules, implicitly denying anything that is not explicitly allowed. If no rule matches the file and user, execution is disallowed.

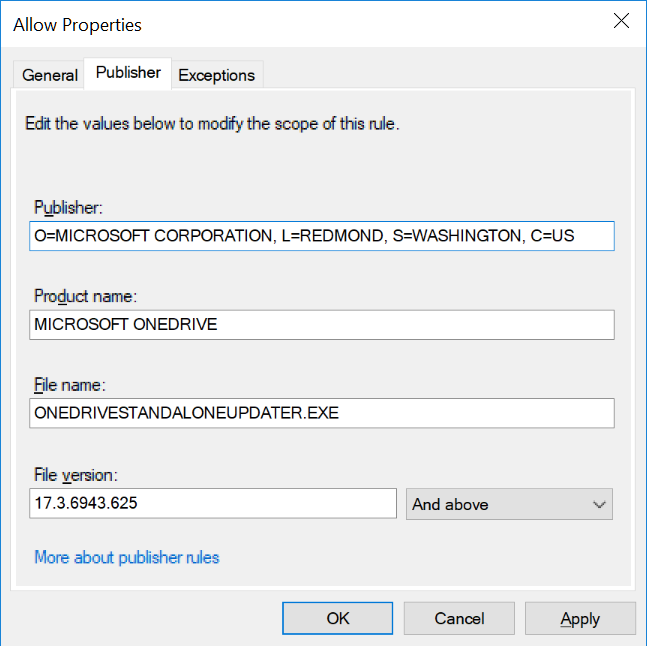
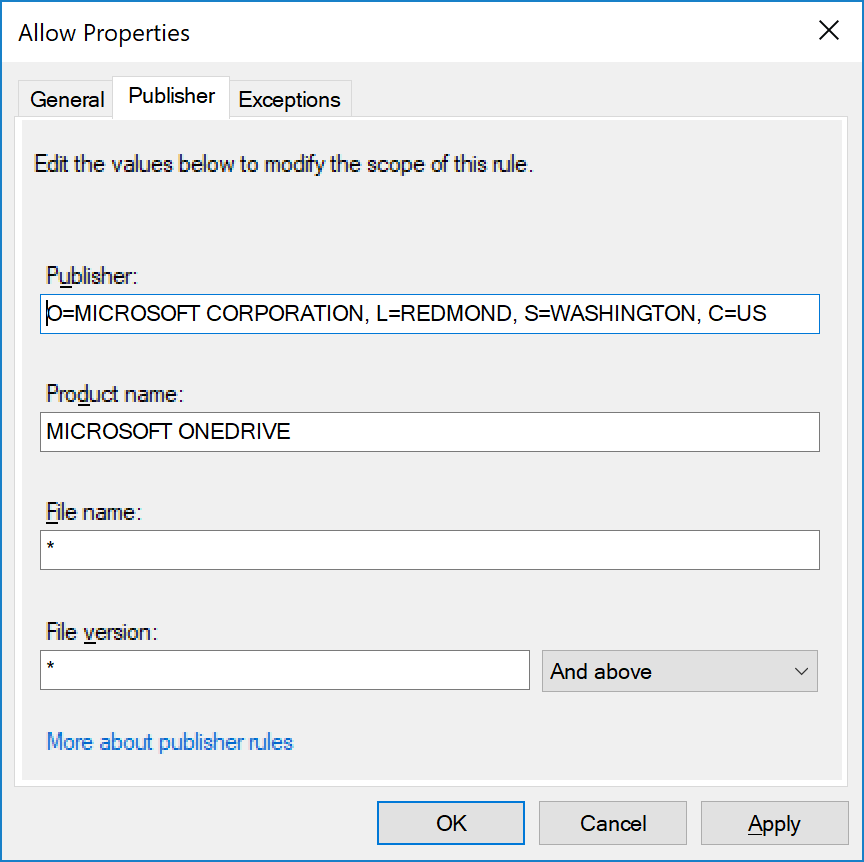
#### Path rules

Path rules are based on files’ locations in the file system and support wild cards. For example, a Path rule that allows execution from %WINDIR%\\* allows execution of everything in the Windows directory and all its subdirectories. Note that although AppLocker uses the same syntax that Cmd.exe uses for environment variables, these are not true environment variables. AppLocker defines a set of names that it recognizes.

🡪 Use Path rules to allow execution only from directories that are read-only to non-administrators. If a Path rule allows execution of a file that a user can modify or replace, that user can run anything of the user’s choice. If any of the subdirectories are user-writable, you should add exceptions to the Path rule for those subdirectories.

#### Publisher rules

Publisher rules are based on files’ digital signatures and optionally on embedded version information including product name, original file name, and file version. A publisher rule can allow (or block) every file signed by a particular publisher or can specify files more granularly by specifying any or all of the product name, original file name, and a specific file version or a minimum or maximum file version. For example, the first screenshot below shows a publisher rule allowing execution of a Microsoft-signed file called OneDriveStandalongUpdate.exe with product name “Microsoft OneDrive” and version number 17.3.6943.625 or higher. The second screenshot defines a less-restrictive rule that allows all Microsoft-signed files with product name “Microsoft OneDrive.” Note that the “file name” attribute comes from the Original File Name attribute in the file’s embedded version information resource[[5]](#footnote-6) and not from the current name in the file system. The embedded Original File Name attribute cannot be modified without invalidating the file’s signature.

🡪 Use Publisher rules primarily to allow execution of approved, signed files that are in user-writable directories. Users cannot modify those files without invalidating the signature, which would cause the rule not to match. A Publisher rule is preferable to a Hash rule (described next) because it can allow for future file updates without having to update the rule.

#### Hash rules

A hash rule allows (or blocks) a specific file based on the SHA256 hash of the file’s content.[[6]](#footnote-7) If the file is modified at all, its hash changes and the rule no longer matches.

🡪 Use hash rules to allow execution of approved, *unsigned* files that are in user-writable directories. Note that if the file is altered at all, a new hash rule must be generated to continue to allow the file to execute. For this reason, Publisher rules are preferable to hash rules.

### “Administrators” and “Everyone”

Although you can specify any user or group in an AppLocker rule, most of the time (and all the time in AaronLocker) rules are only ever applied to “Administrators” or “Everyone” (i.e., everyone else – non-administrators).

Unfortunately, AppLocker is inconsistent about its interpretation of “Administrators.” With User Account Control (UAC) in its recommended configuration, members of the Administrators group run as “protected administrator,” with most programs running with non-administrative rights unless the admin explicitly chooses to “run as administrator.” We have called these two states “unelevated” and “elevated.” When evaluating an executable rule, AppLocker always considers a “protected administrator” to be a member of Administrators, even when running a program unelevated. However, when evaluating rules in the other rule collections, AppLocker considers the user to be a member of Administrators only when running elevated. As a result, consistent executable and DLL rules can lead to inconsistent results, where an executable is allowed to execute but cannot load DLLs from the same directory.

My recommendation therefore is not to use “protected administrator” when using AppLocker, always log on as a non-administrator, and use a separate account for administrative operations.

### AppLocker GUI, default/generated rules and their limitations

The AppLocker GUI includes several options to assist with rule creation. In addition to the Create New Rule wizards, each collection offers options to Create Default Rules or Automatically Generate Rules.

The GUI’s “default rules” provide a good start. They allow non-administrators to run any executables, DLLs, and scripts in the Windows and Program Files directories using Path rules; any signed packaged apps; and any Windows installer files found in the C:\Windows\Installer directory along with any signed Windows installer files. The default rules allow administrators to run everything. The two main problems with the default rules are that 1) the Path rules for the Windows and Program Files directories do not include any exceptions for the many user-writable subdirectories such as C:\Windows\Temp or for executables that often provide whitelisting bypasses such as Mshta.exe; and 2) it makes no provision for code that must be allowed from other locations including the user profile and domain controller shares. For example, OneDrive runs its code from the user profile, and logon scripts typically run from domain controller shares.

The Automatically Generate Rules option is interesting but ultimately falls short of useful. For the Executable, Windows Installer, and Script rule collections, it searches the directory hierarchy you specify (Program Files by default) and generates rules for each file it finds: publisher rules for signed files and either hash rules or path rules (your choice) for unsigned files. It also offers the option to fold as many overlapping rules together to reduce their total number.

There are several flaws in the Automatically-Generate options:

* It offers to generate rules for all the files in the Program Files directory, and then recommends adding the default rules. The default rules allow execution of everything in the Program Files directories, so all the additional rules are redundant.
* The executable-rules scan doesn’t recognize signatures on the packaged apps under Program Files\WindowsApps and generates lots of redundant hash or path rules for each of those files.
* The “Automatically Generate Rules” option is not offered for the DLL Rules collection.
* You always end up with a huge number of rules which become difficult to maintain, as it’s difficult to trace rules back to their original files.

## Intro to AaronLocker

AaronLocker[[7]](#footnote-8) is a set of PowerShell scripts that automates and simplifies the creation and maintenance of robust AppLocker whitelisting policies that address all the limitations described above, and more. You create policies by running scripts, which guarantees consistent and repeatable results. You can easily customize rules with simple text-file edits. AaronLocker creates identical audit-mode and enforce-mode policies at the same time, along with Microsoft Excel spreadsheet documentation. AaronLocker scripts handle the capture and analysis of AppLocker events on managed systems, which simplifies both policy maintenance and the detection of anomalous events.

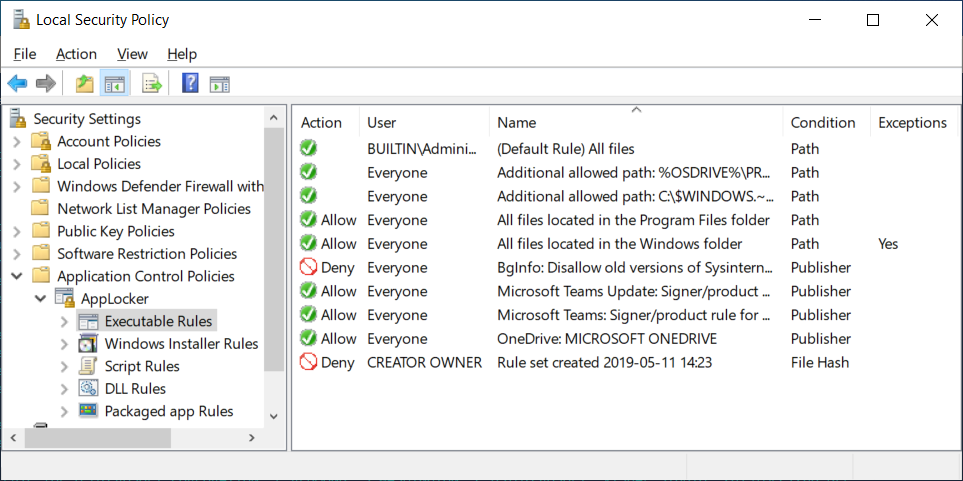
The policy-creation script:

* After scanning for user-writable subdirectories, allows non-administrative execution from the Windows and Program Files directories except for those unsafe locations;
* Blocks well-known whitelisting bypasses such as Mshta.exe and PowerShell versions below 5.0[[8]](#footnote-9);
* Blocks rarely-used but potentially problematic built-in programs such as Cipher.exe (which has been used by ransomware to encrypt files);
* Allows execution from domain controller shares on domain-joined systems;
* Incorporates customizations that you specify;
* Includes rich information in each rule’s name and description so you can trace a rule back to its origins.
* Includes an inert rule that contains the date and time the rule set was generated to help differentiate policy versions. (See screenshot below.) You can retrieve this timestamp from the policy even after it has been imported into Group Policy.[[9]](#footnote-10)

Customizations that you can easily incorporate include:

* “Safe” paths in which to allow execution. (“Safe” paths are directories in which non-administrative users have read-only access.)
* “Unsafe” (user-writable) paths to analyze for files to allow, using publisher or hash rules;
* Trusted publishers/signers to allow, optionally restricting to specific product names, file names, and/or file versions;
* File hashes to trust – typically retrieved from captured event data;
* AppLocker policy fragments and other data captured during earlier scans on other computers.

AaronLocker can be implemented on single Windows systems one at a time using local policy, or deployed to large organizations using Active Directory Group Policy. AppLocker event data can be analyzed locally or forwarded to a central Windows Event Collector for aggregated analysis.



# Part II – AaronLocker Operations Guide

Part II of this document is a step-by-step description of how to implement AaronLocker.

## Getting started – building the first rule sets

The first step with AaronLocker implementation is to build the initial audit-mode and enforce-mode policies. Once those are built then they can be applied, evaluated, and adjusted to meet needs.

### Setting up

The target systems where policy will be applied merely need to be supported editions of Windows that offer AppLocker. If users on the target systems need to be able to run PowerShell, then PowerShell version 5.1 must be installed. AaronLocker blocks the use of PowerShell versions earlier than 5.0.[[10]](#footnote-11) Note that target users must not be in the Administrators group on their computers. The rest of the setup instructions here apply only to the system with which you will build AaronLocker policies.

Start by configuring a Windows system that is representative of target systems that you intend to manage, with as many of your end user programs installed as feasible. The Windows version must of course be supported and be an edition that supports AppLocker. Although not required, some AaronLocker features use Microsoft Excel. To use those features, Excel must be installed.

The system must have PowerShell v5.1, which is installed via the Windows Management Framework v5.1. Script execution must be enabled. To do this, run PowerShell as administrator and run the following command:

Set-ExecutionPolicy RemoteSigned -Force

You will also need Sysinternals AccessChk[[11]](#footnote-12) on the machine on which you create the initial policy. AaronLocker uses AccessChk to determine whether directories are user-writable.

Expand the AaronLocker .zip package to your local file system. If you use separate administrative and non-administrative accounts, you should expand it into a directory that is accessible to both for reading and writing, such as a custom “AaronLocker” directory under the root of C:\, or under C:\Users\Public\Documents. Make sure to keep the .zip package’s directory structure intact. To avoid warnings about potentially unsafe downloaded content, you should “unblock” the .zip file before expanding its contents, using the Unblock-File cmdlet, the file’s Properties dialog in Explorer, or the Sysinternals Streams.exe utility.

Finally, ensure that AccessChk.exe is either in a directory specified in the PATH environment variable, or in the same directory with the main AaronLocker scripts.

### AaronLocker contents

AaronLocker’s seven main scripts are in the package’s root directory and can be categorized as follows:

|  |  |  |
| --- | --- | --- |
| **Rule / Policy generation** | | |
|  | Create-Policies.ps1 | Builds comprehensive and robust AppLocker "audit" and "enforce" rules to mitigate against users running unauthorized software, customizable through simple text files.  *See sample output in the Samples.zip, SampleOutput-CreatePolicies folder.* |
|  | Scan-Directories.ps1 | Scans directories to identify files that might need additional AppLocker rules. |
| **Rule / Policy analysis** | | |
|  | ExportPolicy-ToExcel.ps1 | Turns AppLocker policy into a more human-readable Excel worksheet.  *See sample output in the Samples.zip, SampleOutput-CreatePolicies folder.* |
|  | Compare-Policies.ps1 | Compares two AppLocker policies. |
| **Event analysis** | | |
|  | Get-AppLockerEvents.ps1 | Retrieves and sorts events from AppLocker logs, removes duplicates, and reports as tab-delimited CSV output, an Excel worksheet, or PowerShell objects. |
|  | Save-WEFEvents.ps1 | Captures forwarded events to CSV files with timestamp embedded in the file name. (Wrapper around Get-AppLockerEvents.ps1; intended to be executed on a Windows Event Collector server.)  *See sample output in the Samples.zip, SampleOutput-SaveWEFEvents folder.* |
|  | Generate-EventWorkbook.ps1 | Produces a multi-tab Excel workbook containing summary and details of AppLocker events to support advanced analysis.  *See sample output in the Samples.zip, SampleOutput-GenerateEventWorkbook folder.* |

AaronLocker has seven subdirectories. Note that some of these start out empty and might not be created when you expand the zip file. They will be created the first time you run Create-Policies.ps1.

|  |  |
| --- | --- |
| **Subdirectory name** | **Description** |
| CustomizationInputs | Contains files to specify policy customizations. These files can be edited with a text editor.[[12]](#footnote-13) |
| LocalConfiguration | Contains scripts that assist when configuring AppLocker through local policy |
| MergeRules-Dynamic | Contains intermediate results. Each time Create-Policies.ps1 runs, it deletes any XML files in this directory and generates new ones with AppLocker policy fragments that it then incorporates into the policies it builds. The files represent rules from scans of unsafe directories and specification of trusted signers and hash rules (all described later). |
| MergeRules-Static | Put AppLocker policy fragment files into this directory to have Create-Policies.ps1 incorporate them into the policies it builds. Create-Policies.ps1 does not create or delete files in this directory. |
| Outputs | Create-Policies.ps1 outputs AppLocker policy XML and Excel files into this directory. The filenames incorporate the current timestamp so that older files are not overwritten by newer ones. |
| ScanResults | Create-Policies.ps1 writes the results of scans for user-writable directories into this directory, along with information about built-in executables to blacklist. |
| Support | Contains supporting scripts and other files that AaronLocker operators usually don’t need to use directly. They are documented in an appendix to this document. |

### Defining and tuning “user-writable”

AaronLocker blocks the execution of files in “unsafe” (i.e., user-writable) subdirectories under the Windows and Program Files directories. The purpose of this iteration is to identify those directories and to tune AaronLocker’s definition of “user-writable” if necessary. From an administrative PowerShell console[[13]](#footnote-14), run the following command:

.\Create-Policies.ps1

Although this will make a complete policy-building pass, the only files we need to inspect at this time are the \*.xml files in the ScanResults subdirectory: Writable\_Full\_PF.xml, Writable\_Full\_PF86.xml, and Writable\_Full\_windir.xml. These three XML files list all the user-writable directories found in the Program Files, Program Files (x86), and Windows directories, respectively, and the non-administrative entities that are granted some type of “write” permission to them. If any of those entities are actually administrative accounts that AaronLocker did not recognize as administrative, the resulting policies might block execution in those directories unnecessarily.

To display a list of the user/group entities named in those XML files, run this PowerShell command:

Get-ChildItem .\ScanResults\\*.xml | foreach {([xml](Get-Content $\_)).root.dir.Grantee} | Sort-Object -Unique

If you are certain that any of the names in the list are administrative users or groups, add them to **CustomizationInputs\KnownAdmins.ps1**, one to a line and in double-quotes. For example:

"DESKTOP-7TPCJ7J\renamedAdmin"  
"CONTOSO\SCCM-Admins"

Make sure not to add any non-administrative users or groups to this file. *If you are uncertain, do not add it to KnownAdmins.ps1*.

After editing KnownAdmins.ps1, run this command to update the ScanResults contents:

.\Create-Policies.ps1 -Rescan

You should periodically perform a similar scan especially after new applications are installed to verify whether there are any new unsafe directories.

### Initial customizations

The next step is to create rules allowing execution of required files outside of the Windows and Program Files directories, and if necessary from unsafe subdirectories in the Windows and Program Files directories. In a perfect world, all the files that anyone would ever need to run would be in “safe” subdirectories of the Windows and Program Files directories. Unfortunately, that is rarely the case. Such files sometimes get placed into the user’s profile, to ProgramData subdirectories, and to non-standard directories created in the root directory of the C: drive. And some installers put files in Program Files subdirectories where they belong but then make them unsafe by granting Everyone “full control” over those subdirectories. Without any customizations, AaronLocker base policy will block these files from running. If non-administrators need to run any of these files, the AppLocker policy will need custom rules.

The important distinction is whether the files are in *safe* or *unsafe* directories. Path rules can be used for files in safe directories; publisher or hash rules must be used for files in unsafe directories. The user profile is always unsafe; non-default C:\ subdirectories are usually unsafe; and ProgramData subdirectories might go either way. The **Scan-Directories.ps1** script described in the next section can help make these determinations.

You specify customizations with text-file edits to the simple PowerShell scripts in the **CustomizationInputs** directory. These scripts each produce text or hash tables. **Create-Policies.ps1** calls each of these scripts and turns their output into corresponding AppLocker rules.

#### Safe paths

To reiterate, a path is “safe” only if it is read-only to non-administrators. Only administrators have the ability to create, modify, or delete files in safe paths.

To create Path rules for safe paths containing files that users need to run, edit **CustomizationInputs\GetSafePathsToAllow.ps1**. This script’s output is merely a list of safe paths, one to a line. To allow execution of files in the C:\ProgramData\App-V directory, which is read-only to non-admins, simply add a line to GetSafePathsToAllow.ps1 so that it outputs that path:

"C:\ProgramData\App-V\\*"

*Specify paths using only local fixed drive letters or UNC paths.* Do not use mapped drive letters or SUBST drive letters, as the user can change their definitions. When you define a rule using UNC paths, it will still work correctly if the program is started with a mapped drive. For example, if X: is mapped to the read-only \\MYSERVER\Apps file share, and you allow execution in \\MYSERVER\Apps\\*, the user will be able to run MyProgram.exe in that share whether it is referenced as \\MYSERVER\Apps\MyProgram.exe or as X:\MyProgram.exe. Similarly, AppLocker does the right thing with SUBSTed drive letters.

GetSafePathsToAllow.ps1 includes code that adds domain controller NETLOGON and SYSVOL shares to the safe-paths list if the computer is joined to an Active Directory domain. This is primarily to allow logon scripts to execute.

#### Unsafe paths

A directory path is “unsafe” if non-administrators are able to create, modify, or delete files in it. If users need to run files in unsafe directories, those files should be specified with Publisher rules if possible, and with Hash rules otherwise.

You can specify unsafe paths that require rules by editing **CustomizationInputs\UnsafePathsToBuildRulesFor.ps1**. This script outputs PowerShell hash tables each specifying a label to document the rules with, along with one or more paths to scan, and optionally how granular any Publisher rules should be. Create-Policies.ps1 inspects all the files in these paths and generates corresponding Publisher rules where possible, and Hash rules otherwise. The main benefit of Publisher rules is that they don’t need to be updated every time files are updated. Hash rules for files need to be updated whenever unsigned files are added or modified.

The comments in the UnsafePathsToBuildRulesFor.ps1 script file show several examples demonstrating different usages and options. This is a typical example:

@{

label = "OneDrive";

paths = "$env:LOCALAPPDATA\Microsoft\OneDrive";

pubruleGranularity = "pubProduct";

}

When Create-Policies.ps1 processes this hash table, it will inspect every executable, DLL, and script in %LOCALAPPDATA%\Microsoft\OneDrive and its subdirectories. For signed files it will create a publisher rule for every publisher/product combination that it finds (with special handling for certain Microsoft-signed files as described below). It will also create a separate Hash rule for any unsigned files, or signed files that do not have version information. Each rule will include “OneDrive” in its name.

You can choose from four levels of Publisher rule granularity, with special handling for Microsoft-signed files:

* **pubOnly** – create rules allowing any files signed by the publishers of the scanned files. For example, if the scan finds 30 files signed by “Contoso” and 20 signed by “Fabrikam,” Create-Policies.ps1 creates two Publisher rules, one allowing any files signed by “Contoso” and one allowing any files signed by “Fabrikam.” The pubOnly option creates the least granular and often the smallest number of rules. It can be the most flexible when a publisher’s product names are inconsistent or absent.
* **pubProduct** – create rules allowing any files with the same publisher and product name attributes of any of the scanned files. For example, if the scan finds 30 files signed by “Contoso” and all of them have the product name “Contoso Software,” Create-Policies.ps1 creates one Publisher rule allowing any files signed by “Contoso” but only if they also have the product name “Contoso Software.” The pubProduct option is more granular than pubOnly and might produce more rules than pubOnly, but typically far fewer than pubProductBinary. It works well when a product’s updates might add new signed files.
* **pubProductBinary** – creates a separate Publisher rule for each signed file, specifying the publisher, the product name, and the internal binary name. This is the default granularity if one isn’t specified.
* **pubProdBinVer** – creates a separate Publisher rule for each signed file, specifying the publisher, the product name, the internal binary name, and the minimum file version allowed. The scanned file itself is allowed and any update with a higher version number. This ensures that older versions of the allowed files are not allowed to run.

Because Microsoft publishes a wide range of products including debuggers and other tools with which one can execute arbitrary code, the minimum granularity for rules created for Microsoft-signed files found in an unsafe-path scan is *pubProduct*; the minimum granularity if the Microsoft product is Windows or Visual Studio is *pubProductBinary*. For example, if you scan a “Contoso Software” directory and specify *pubOnly* and the directory contains Microsoft-signed Visual Studio redistributable files, Create-Policies.ps1 will create a single Publisher rule allowing all files signed by Contoso and separate Publisher rules for each Visual Studio file scanned.

Note for later: when using environment variables pointing to user-profile subdirectories, you might need to use the “-ForUser” parameter with Create-Policies.ps1, described in the “Build AaronLocker policies…” section later.

#### Trusted Signers

If you know that you want to trust everything signed by a specific publisher, or a particular product by that publisher, edit **CustomizationInputs\TrustedSigners.ps1**. This script outputs hash tables each specifying a label to document the rules with, along with information to plug into Publisher rules. You can limit the trust just to files in a specific rule collection (e.g., trust all Microsoft-signed scripts), or restrict to a specific version of a specific file. And instead of specifying the literal Publisher rule information, you can reference a file and AaronLocker will use the signature information from that file to generate the Publisher rule. See the TrustedSigners.ps1 script’s comments for the details.

By default, TrustedSigners.ps1 trusts Microsoft-signed script files, MSI files, Teams product executables, and MSVC/MFC redistributable DLLs (listed in TrustedSigners-MsvcMfc.ps1). Windows’ built-in troubleshooting features often involve running Microsoft-signed scripts in the user’s profile. Note that trusting all Microsoft-signed files or all Microsoft-signed EXE files is an overly broad whitelisting strategy. Create-Policies.ps1 will issue a warning if you set a rule like that, as it would allow non-administrators to run a huge variety of programs from the user profile, removable media, etc.

TrustedSigners.ps1 also includes entries (commented-out by default) to support Google Chrome. Chrome runs some code in the user profile even when Chrome is installed to Program Files. Uncomment this block of entries to enable those components to run.

To build rules specified by paths in the UnsafePathsToBuildRulesFor.ps1 script, Create-Policies.ps1 needs to see the actual files to base rules upon. By contrast, TrustedSigners.ps1 does not. This can be useful when modifying rules based on data from Windows Event Collector, where you have forwarded event data but not direct access to the files involved.

#### Customizing ExeFilesToBlacklist

AaronLocker blocks the execution of built-in Windows programs that are rarely needed by non-administrative users but that have been documented as whitelisting bypasses or that can cause other problems. For example, Cipher.exe is a command-line tool primarily for managing the NTFS Encrypting File System (EFS) feature, but it has been used by ransomware to encrypt files. If you need to allow some of these programs, you should comment them out in **CustomizationInputs\GetExeFilesToBlacklist.ps1**.

The script outputs the full paths of each program under the Windows directory to block, one to a line.

#### Merge additional policy fragments

The **MergeRules-Static** directory can contain XML files specifying AppLocker policy fragments that Create-Policies.ps1 merges into your full policy. The default AaronLocker package includes XML files with rules that cover the Microsoft OneDrive files that shipped in Windows 10 versions 1607, 1803, and 1809. Incorporating one or more of these files ensures that if you enforce AaronLocker policy on a fresh Windows 10 v1607, v1803, or v1809 system, OneDrive will be able to run enough to update itself to the current version. The default package also includes a file that disallows older versions of Sysinternals BgInfo.exe that were not AppLocker-aware and allowed execution of unapproved VBScript files. And the default package includes a file allowing Google Chrome’s signed Flash Player DLL to load. That DLL’s version resource has an anomaly that creates a minor incompatibility with AppLocker’s evaluations. If you don’t need these, you can delete them or change the extensions to something other than “.xml.”

A later section of this document describes how you can add files to this set.

### Getting the data for first customizations

AaronLocker’s **Scan-Directories.ps1** script can determine whether directories contain files that might be subject to AppLocker rules (i.e., executable files, DLLs, scripts, and installer files) and whether those directories are safe. This can aid in the decision whether those files actually need to be allowed to run, and provides the data needed for the customization edits described in the “First customizations” section above. Scan-Directories.ps1 produces tab-delimited CSV, or optionally a formatted Excel spreadsheet ready for sorting and filtering.

To scan directories for files of interest (executables, scripts, etc.), use one or more of the parameters described in the table below. To output to Excel rather than simple CSV, add the ‑Excel switch to the command line.

|  |  |  |
| --- | --- | --- |
| **Directories to search** | | |
|  | -WritableWindir | Searches the writable subdirectories of the %windir% directory, based on the results of the last scan performed by Create-Policies.ps1 |
|  | -WritablePF | Searches the writable subdirectories of the %ProgramFiles% directories, based on the results of the last scan performed by Create-Policies.ps1 |
|  | -SearchProgramData | Searches the %ProgramData% directory hierarchy |
|  | -SearchOneUserProfile | Searches the current user's profile directory |
|  | -SearchAllUserProfiles | Searches all user profile directories (i.e., everything under C:\Users) |
|  | -SearchNonDefaultRootDirs | Searches all non-standard directories in the %SystemRoot% root directory. These directories often contain LOB applications. |
|  | -DirsToSearch *paths* | Search one or more caller-specified, comma-separated directory paths. For example: Scan-Directories.ps1 ‑DirsToSearch C:\Temp, C:\Apps |

You can combine as many of these parameters as you want. For example:

Scan-Directories.ps1 ‑WritableWindir ‑WritablePF ‑SearchProgramData ‑DirsToSearch C:\Temp ‑Excel

Scan-Directories.ps1 also offers a separate command line switch, ‑FindNonDefaultRootDirs, to identify any custom, non-standard directories in the C: root directory on the current system. If any are identified, the script outputs them one to a line. You can scan those directories for files of interest with the -SearchNonDefaultRootDirs switch:

Scan-Directories.ps1 ‑SearchNonDefaultRootDirs -Excel

For each file of interest, Scan-Directories.ps1 outputs the columns listed in the table below. The third column in the table names the scripts that this data can feed into. (Those scripts are described in the “First customizations” section earlier.)

|  |  |  |
| --- | --- | --- |
| **Column name** | **Description** | **Feeds customization of…** |
| IsSafeDir | “SafeDir” or “UnsafeDir”, depending on whether the file’s parent directory is user-writable. | GetSafePathsToAllow.ps1, UnsafePathsToBuildRulesFor.ps1 |
| File type | From AppLocker’s perspective: EXE, DLL, Script, or MSI | TrustedSigners.ps1 |
| File extension | The file’s extension (enabling sorting and filtering and to help distinguish between EXE and DLL files). |  |
| File name | The file name without the directory path | HashRuleData.ps1 |
| File path | The file’s full path information | HashRuleData.ps1 |
| Parent directory | The file’s parent directory (see also the ‑DirectoryNamesOnly switch, described below) | GetSafePathsToAllow.ps1, UnsafePathsToBuildRulesFor.ps1 |
| Publisher name | If signed, the publisher name | TrustedSigners.ps1 |
| Product name | If signed, the product name | TrustedSigners.ps1 |
| Binary name | If signed, the “OriginalName” field taken from the file’s version resource | TrustedSigners.ps1 |
| Version | If signed, the binary file version taken from the file’s version resource | TrustedSigners.ps1 |
| Hash | The file’s hash | HashRuleData.ps1 |
| CreationTime | When the file was created, according to the file system |  |
| LastAccessTime | When the file was last accessed, according to the file system (if the file is not writable, this timestamp doesn’t get updated) |  |
| LastWriteTime | When the file was last modified, according to the file system |  |
| File size | The length of the file | HashRuleData.ps1 |

You can simplify the Scan-Directories.ps1 output with the ‑DirectoryNamesOnly switch, which outputs only the *IsSafeDir* and *Parent directory* columns. With this switch, each directory is listed at most once, rather than once for each file of interest it contains. The directory name can be used in GetSafePathsToAllow.ps1 or UnsafePathsToBuildRulesFor.ps1, depending on its safety.

For example, too many app installers change file system permissions so that the app can write data into its installation directory or a subdirectory. AaronLocker blocks execution from those locations. That’s not a problem if the subdirectory is only for data files, but if it contains executable or script files, you’ll need more customization. Scan-Directories.ps1 -WritablePF lists all the files in those directories that would be blocked by AppLocker. Scan-Directories.ps1 -WritablePF -DirectoryNamesOnly just lists the directories they’re in. You can use those directory names in UnsafePathsToBuildRulesFor.ps1.

Note: I strongly recommended to ensure that any Unicode and other special characters in the event data such as ® and ™ survive the pipeline by setting the OutputEncoding preference variable like this before calling Scan-Directories:

$OutputEncoding = [System.Text.ASCIIEncoding]::Unicode  
You can also do this using the Set-OutputEncodingToUnicode.ps1 script in the Support directory.

### Build AaronLocker policies for pilot testing

With these customizations in place, you are ready to build your first policy to apply to systems. With the results of the earlier scans still in the ScanResults subdirectory, you can perform this step with either an administrative or non-administrative account, as those scans do not need to be repeated. If you use a non-administrative account and the unsafe paths you need to scan are in that account’s user profile, run this command:

.\Create-Policies.ps1 -Excel

If you use an administrative account and some of the unsafe paths to scan are in another account’s user profile, run this command, replacing *username* with the name of the other account:

.\Create-Policies.ps1 -ForUser *username* -Excel

The *username* you supply replaces the current account name in file paths when scanning user-profile paths.

AppLocker uses XML to represent policies for import and export. Create‑Policies.ps1 creates two XML files in the Outputs subdirectory representing the full AppLocker policy incorporating all your customizations. The file names also include the date and time so that older files are not overwritten. For example, these audit and enforce policies were generated on June 10, 2018, at 9:43pm:

.\Outputs\AppLockerRules-20180610-2143-Audit.xml  
.\Outputs\AppLockerRules-20180610-2143-Enforce.xml

With the -Excel switch, Create-Policies.ps1 also generates a formatted Excel spreadsheet documenting each policy in a much more readable format that can also be sorted and filtered. The file names are the same as for the XML files but with the .xlsx extension. If you don’t have Microsoft Excel installed, don’t use the ‑Excel switch. If it is installed but hasn’t been used recently or at all, make sure to run it a few times to make sure there aren’t any first-run or “here’s what’s new” dialogs that can interfere with automation.

You can see sample output from Create-Policies.ps1 including Excel output in the Samples.zip, SampleOutput-CreatePolicies folder.

### Next steps

You now have an XML file representing a robust set of AppLocker rules in audit mode, and another XML with the exact same set of rules but in enforce mode. Next steps: apply the policy to one or more target systems, evaluate the results, and tune the policy if needed. Your primary choices are to deploy to your organization through Active Directory Group Policy, or to deploy to individual systems using local policy. In the former case, I recommend also setting up a Windows Event Collector server and configuring Windows Event Forwarding on the managed systems. Details for both cases are in the coming sections.

### Tip: moving AaronLocker policy management to another computer

Certain portions of an AaronLocker-generated policy depend on data captured by Create‑Policies.ps1 from files on the local computer. Once that information has been captured, though, you can move the captured data to another computer and perform updates on the policy there – even on another version of Windows. In one real-world example, I built a policy for a Windows 8.1 tablet. I have since been performing updates to that policy on a Windows 10 desktop system but continuing to use the information captured on the Windows 8.1 system.

Copy the AaronLocker package contents including all the subdirectories containing customized files and intermediate results from the original computer to the new computer. On the new computer, move all the files in **MergeRules‑Dynamic** *except for TrustedSigners.xml and ExtraHashRules.xml*, if present, to the **MergeRules‑Static** directory. Then comment out all the corresponding hash tables in the **UnsafePathsToBuildRulesFor.ps1** script. This keeps Create‑Policies.ps1 from scanning those unsafe paths on the new system while retaining the rules generated on the old system. (The content in TrustedSigners.xml and ExtraHashRules.xml is built from content in TrustedSigners.ps1 and HashRuleData.ps1 without scanning files.)

## Applying AaronLocker through local policy

The scripts in the LocalConfiguration subdirectory assist with configuring AppLocker on individual computers through local policy. Each requires administrative rights.

* **ConfigureForAppLocker.ps1** Performs basic one-time configuration changes to support AppLocker use. It configures the Application Identity (AppIdSvc) service for automatic start, starts the service, and configures the AppLocker event log sizes to 1GB each. The logs’ default 1MB sizes are small and overrun quickly. Changing the log sizes isn’t usually necessary when Windows Event Forwarding is configured, but is beneficial for standalone, single-system testing. (1GB is rather a lot – half of that would probably be fine.)
* **ApplyPolicyToLocalGPO.ps1** Applies the most recent AaronLocker-generated policy to local Group Policy. Applies the enforce-mode policy by default. To apply the audit-mode policy instead, add the **‑AuditOnly** switch to the command line.
* **ClearLocalAppLockerPolicy.ps1** Reverts the local AppLocker policy to “not configured.”
* **ClearAppLockerLogs.ps1** Clears all events from the local AppLocker event logs.

See “Tuning the rule set” for next steps after applying AppLocker policy.

## Deploying AaronLocker through Active Directory and Group Policy Objects

When introducing AppLocker to an organization, start with a relatively small pilot in audit mode so you can evaluate the effects without interfering with users’ work. Adjust the rules to address issues and expand the pilot, increasing the number of audit-mode systems while transitioning some to enforce mode. Continue that cycle, adjusting rules to minimize impact, moving the entire organization into audit mode and transitioning more pilot systems into enforce mode. Eventually bring the entire organization into enforce mode, while also retaining the ability to create exceptions as needed and either move specific computers into audit mode or exempt them from domain-managed AppLocker policy entirely.

To manage this process, we recommend establishing two GPOs – one for audit mode and one for enforce mode – and three security groups that will be used to determine which computers receive those policies.

### Security groups

Create three domain-local security groups: “AppLocker-Audit,” “AppLocker-Enforce,” and “AppLocker-Exempt”. During different phases, adding computers to these groups controls whether they are in audit mode, in enforce mode, or are exempt from domain-managed AppLocker policy. You’ll adjust the GPOs’ security filtering to transition through the pilot, tuning, and final production phases. Ideally, in the final state all three security groups will be empty and the AppLocker enforce policy will be applied to all workstations enterprise wide.

If AppLocker enforce-mode policy interferes with a user’s ability to get work done, that computer can be moved temporarily to the Audit group until the problem is resolved. Because Active Directory AppLocker policy overrides any local AppLocker policy, membership in the Exempt group enables quick testing with local AppLocker policy without having to involve AD management.

### Group Policies

In this section, you’ll create two GPOs: AppLocker-Audit and AppLocker-Enforce. The settings in each GPO will include:

* Your audit-mode or enforce-mode AppLocker policy;
* Service configuration making the Application Identity (AppIdSvc) and Windows Remote Management (WinRM) services Automatic start;
* Windows Event Forwarding configuration to forward AppLocker events to the Windows event collector server, which is described later in this section.

You will also configure Security Filtering on these GPOs.

#### Configuring GPO settings

In the Group Policy Management Console (GPMC), create and edit a GPO called “AppLocker-Audit” and another called “AppLocker-Enforce.” In the Group Policy editor, navigate to Computer Configuration, Policies, Windows Settings, Security Settings, Application Control Policies, AppLocker. Right-click on the AppLocker icon in the console tree, choose “Import Policy…” and pick the audit-mode or enforce-mode AppLocker policy XML you created.

To configure the Application Identity service, navigate to System Services which is also under Security Settings. Double-click on “Application Identity” in the pane to the right, define the policy setting and choose Automatic startup mode. Do not click the “Edit Security” button. Then do the same for the “Windows Remote Management (WS-Management)” service.

Finally, to configure Windows Event Forwarding, navigate to Computer Configuration, Policies, Administrative Templates, Windows Components, Event Forwarding, and edit the “Configure target Subscription Manager” setting. Select Enabled, click the Show button, and enter this text in the SubscriptionManagers list:

Server=http://*WECServerFQDN*:5985/wsman/SubscriptionManager/WEC,Refresh=60

Replace “*WECServerFQDN*” with the fully-qualified domain name of the server you’ll configure as your Windows Event Collector server. Details about setting up that server are in an upcoming section.

#### Configuring GPO Security Filtering

In the early stages, computers will get AppLocker policy only if they are in the Audit or Enforce groups. When the pilot has expanded broadly, all computers will receive Audit policy unless they are in the Enforce or Exempt groups. In production deployment, all computers will receive the Enforce policy unless they are in the Audit or Exempt groups. All this is accomplished by configuring Security Filtering on the two GPOs.

Note that no computer should be in more than one of the AppLocker security groups. Computers don’t need to be in any of the groups, but they should not be in two or more groups.

A Group Policy Object can be applied to members of a security group only if the group is granted the Read and Apply permissions to the GPO. Explicitly denying a group the Apply right takes precedence over another entry granting Read + Apply. The Audit group is always granted Read + Apply to the Audit GPO, and the Enforce group is always granted Read + Apply to the Enforce GPO. The Enforce and Exempt groups are always granted Read + *Deny* Apply to the Audit GPO, and the Audit and Exempt groups are always granted Read + *Deny* Apply to the Enforce GPO. The permissions granted to the Authenticated Users group are the only ones that change as you move from the initial pilot phase to the broad pilot phase and finally to production deployment.

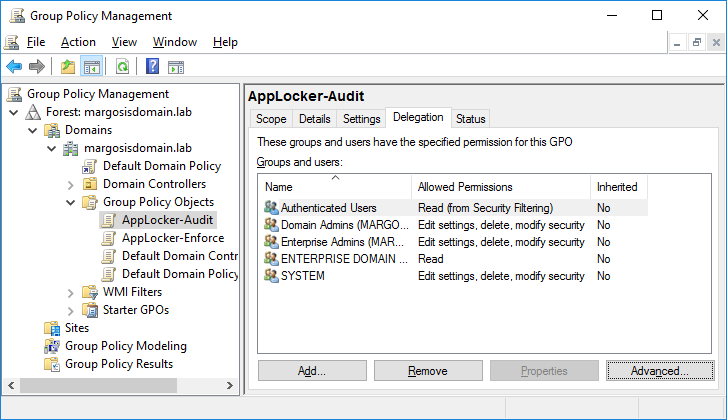
By default, a newly-created GPO grants Read and Apply to Authenticated Users (AU), a group that includes all domain computers. For the initial pilot, we want the GPOs to be applied only to members of the Audit or Enforce groups, so we will remove AU’s Apply permission. AU must always have at least the Read permission to both GPOs. Granting AU both Read and Apply to one of the GPOs makes that GPO the default for all domain computers.

This table summarizes how to configure Security Filtering on the AppLocker-Audit and AppLocker-Enforce GPOs, starting with the initial pilot phase, then as you move to a broad pilot, and finally to full production AppLocker enforcement. Note that the permissions for “Authenticated Users” are the only ones that change as you move through the phases, and that AU must always have at least “Read” permission to both GPOs.

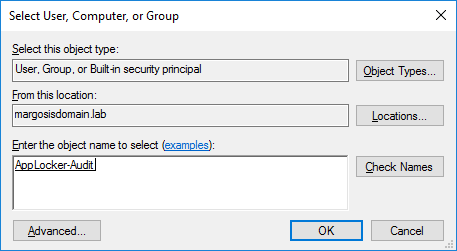
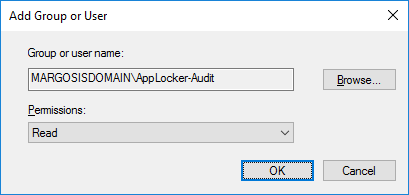
|  |  |  |
| --- | --- | --- |
|  | **Security Filtering** | |
| **Phase** | **AppLocker-Audit GPO** | **AppLocker-Enforce GPO** |
| Initial pilot phase | Authenticated Users (Read)  Audit group: Read + Apply  Enforce Group: Read + *Deny* Apply Exempt group: Read + *Deny* Apply | Authenticated Users (Read)  Enforce group: Read + Apply  Audit group: Read + *Deny* Apply Exempt group: Read + *Deny* Apply |
| Broad pilot | Authenticated Users (Read **+ Apply**)  Audit group: Read + Apply  Enforce Group: Read + *Deny* Apply Exempt group: Read + *Deny* Apply | Authenticated Users (Read)  Enforce group: Read + Apply  Audit group: Read + *Deny* Apply Exempt group: Read + *Deny* Apply |
| Production | Authenticated Users (**Read**)  Audit group: Read + Apply  Enforce Group: Read + *Deny* Apply Exempt group: Read + *Deny* Apply | Authenticated Users (Read **+ Apply**)  Enforce group: Read + Apply  Audit group: Read + *Deny* Apply Exempt group: Read + *Deny* Apply |

##### Security Filtering for the initial pilot phase

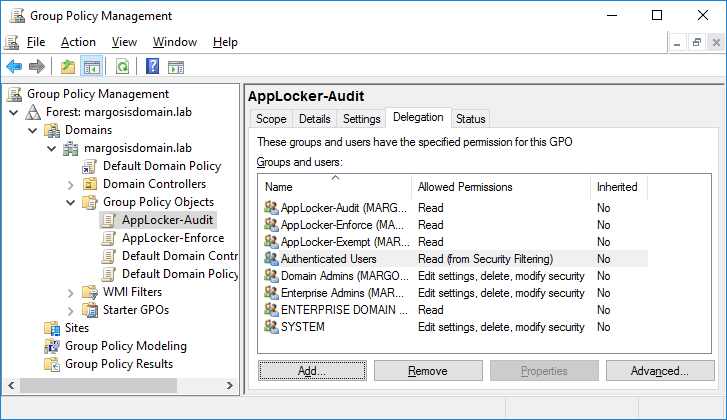
In the GPMC, click on the AppLocker-Audit GPO in the console tree. The Scope tab in the right pane of the GPMC includes a Security Filtering section, but we need a more advanced interface to be able to configure “Apply” permissions. Click on the Delegation tab – see screenshot, below:



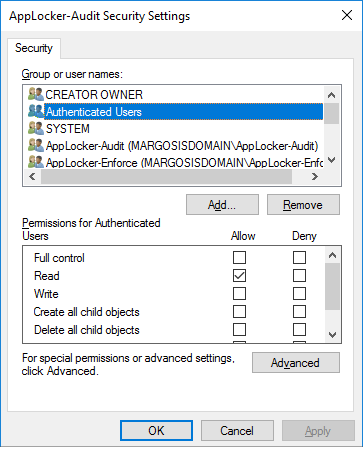
Start by granting the AppLocker-Audit, AppLocker-Enforce, and AppLocker-Exempt groups “Read” permission. To do this, click the Add button, enter “AppLocker-Audit” (without the quotes), Check Names, OK, and then OK again (see screenshots). Do the same for the AppLocker-Enforce and AppLocker-Exempt groups.

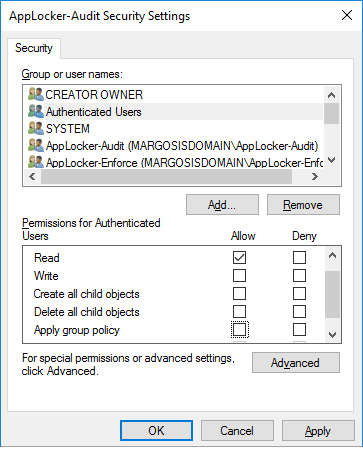
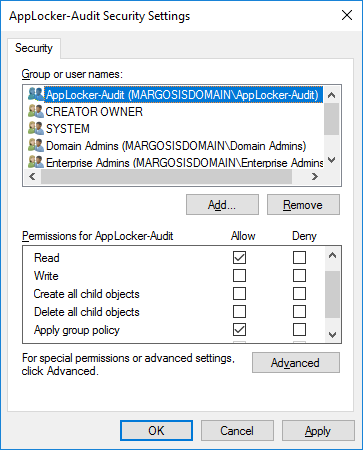
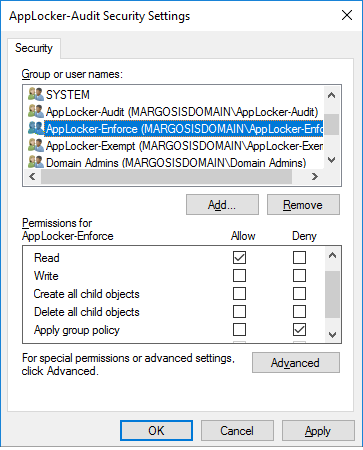
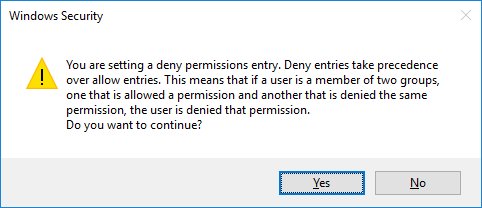
You should now have granted Read to all three custom groups, as shown here:



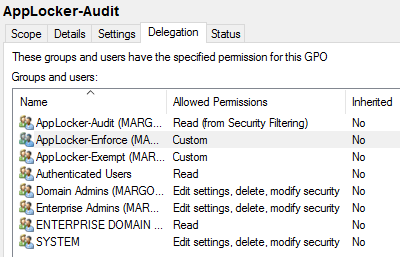
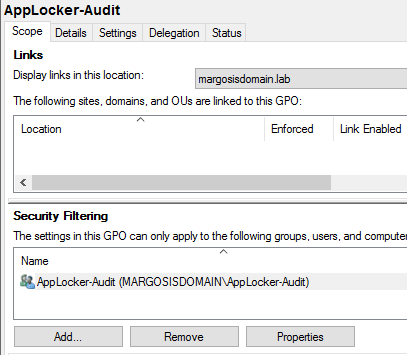
Click the Advanced button to open the GPO’s ACL editor, shown here:



To configure permissions for the initial pilot phase:

* Select “Authenticated Users,” verify that the Allow box is checked for Read, then *uncheck* the Allow box for the “Apply group policy” permission, as shown:  
  
* Select the AppLocker-Audit group, verify that the Allow box is checked for Read, then *check* the Allow box for the “Apply group policy” permission, as shown:  
  
* Select the AppLocker-Enforce group, verify that the Allow box is checked for Read, then check the *Deny* box for “Apply group policy,” as shown:  
  
* Select the AppLocker-Exempt group and configured it the same way as AppLocker-Enforce (allow Read, *deny* Apply).
* Click OK and confirm the setting of a “deny” entry if prompted.  
  

Note that in the Delegation tab, “Read + Apply” is reported as “Read (from Security Filtering)” and will be listed in the Scope tab’s Security Filtering section; “Read + *Deny* Apply” is shown as “Custom” on the Delegation tab and is not listed on the Scope tab. For the initial pilot phase, only AppLocker-Audit should be listed in the Security Filtering section. See screenshots:

Now repeat all of the steps in this section with the AppLocker-Enforce GPO, but granting Apply to the Enforce group and denying Apply to the Audit group. To summarize:

* Grant the AppLocker-Audit, AppLocker-Enforce, and AppLocker-Exempt groups Read permission;
* Click Advanced;
* Select Authenticated Users and uncheck the Apply group policy permission;
* Select AppLocker-Audit and check the *Deny* Apply group policy permission;
* Select AppLocker-Enforce and check the Allow Apply group policy permission;
* Select AppLocker-Exempt and check the *Deny* Apply group policy permission;
* Verify that the Security Filtering section on the Scope tab lists only AppLocker-Enforce.

##### Security Filtering for the broad pilot and production phases

In the initial pilot phase, computers that are not explicitly in one of the AppLocker security groups do not have AppLocker policies applied through Group Policy. In the broad pilot phase, computers that aren’t in one of the groups have the AppLocker-Audit GPO applied. In final production deployment, computers that aren’t in one of the groups have the AppLocker-Enforce GPO applied. To move from one phase to the next, simply change the GPOs’ permissions for the Authenticated Users group, as shown in this table:

|  |  |  |
| --- | --- | --- |
|  | **GPO permissions for Authenticated Users** | |
| **Phase** | **AppLocker-Audit GPO** | **AppLocker-Enforce GPO** |
| Initial pilot phase | Read | Read |
| Broad pilot | Read + Apply | Read |
| Production | Read | Read + Apply |

Note that the two GPOs should never both have Read + Apply at the same time. In the initial pilot, neither have both permissions; in later phases only one of them does at a time.

When you grant Read + Apply to AU, the Security Filtering section adds AU to the Security Filtering list, as shown here:



### Configuring the Windows Event Collector

Designate a Windows Server to be the Event Collector to receive AppLocker warning and error events from all computers receiving the AppLocker policy. Ensure that its fully-qualified domain name (FQDN) is specified in the Event Forwarding settings for the AppLocker-Audit and AppLocker-Enforce GPOs, as described in the “Configuring GPO settings” section.

To configure the server manually, follow these steps:[[14]](#footnote-15)

* Run “winrm quickconfig” or perform equivalent steps (configure the WinRM service for automatic start, create a listener, and enable a firewall exception to allow inbound connections).
* Open Event Viewer and select the Subscriptions node. If the Windows Event Collector Service is not already configured, the Event Viewer will offer to start the service and configure it for Automatic start. Click Yes.
* Right-click the Subscriptions node and select “Create Subscription…” to open the Subscription Properties dialog.
* Enter a name such as “AppLocker Events.” Keep the destination log at the default “Forwarded Events.”
* Select “Source computer initiated” and click the “Select Computer Groups” button.
* In the Computer Groups dialog, click “Add Domain Computers…,” and in the Select Computer or Group dialog, enter “Domain Computers” and click OK. Click OK to close the Computer Groups dialog.
* Click Select Events to open the Query Filter dialog. Check the Warning and Error boxes. Make sure “By log” is selected and click the “Event logs” dropdown. Navigate to Applications and Services Logs, Microsoft, Windows, AppLocker, and check both “EXE and DLL” and “MSI and Script.” Click OK.
* Click Advanced, and under “Event Delivery Optimization,” select Minimize Latency. Click OK.
* Click OK to close the Subscription Properties dialog.

## Evaluating results and tuning the rule set

With users now operating with AppLocker rules either in audit mode or enforce mode, the next step is to determine those rules’ effects and whether you need to adjust those rules – or the blocked programs themselves. The answers to “What is getting blocked?” are in Windows event logs. This section describes how to get that information and what to do with it.

AppLocker writes an event to a Windows event log every time it evaluates whether a file should be allowed to run. Each time a file is about to be executed and the file type’s rule collection is enabled, AppLocker determines whether the collection’s rules allow the file to run. If they do, AppLocker writes an Information event to a Windows event log. If they don’t, then if the rule collection is configured to “enforce rules” AppLocker writes an error to the event log and prevents the file’s execution; if the collection is configured for “audit only” AppLocker writes a warning to the event log. AppLocker writes to the “Microsoft-Windows-AppLocker/EXE and DLL” log for events involving EXE or DLL files; it writes to the “Microsoft-Windows-AppLocker/MSI and Script” log for events involving MSI or script files.[[15]](#footnote-16)

A big problem, though, is that raw data from AppLocker event logs is not exactly easy to work with, particularly in a large environment or when you have a lot of data. AaronLocker includes three scripts that vastly simplify this work: **Get-AppLockerEvents.ps1**, **Save-WEFEvents.ps1**, and **Generate-EventWorkbook.ps1**.

**Get-AppLockerEvents.ps1** retrieves AppLocker event data from local or remote live logs, local or remote event collectors, or saved log files; sorts, filters, and synthesizes data; and outputs its results to the PowerShell pipeline as tab-delimited CSV, PowerShell objects, or a formatted Excel spreadsheet. CSV results can be written to a file or converted into PowerShell objects with the ConvertFrom-Csv cmdlet for further processing. The script’s rich set of parameters enable you to specify filters on event types, date and time ranges, and which fields to include in the output.

**Save-WEFEvents.ps1** is a wrapper around Get-AppLockerEvents.ps1 that’s designed specifically for use on a Windows Event Collector server. It retrieves events from the Forwarded Events log (rather than from the default AppLocker logs) and writes the results to a Unicode tab-delimited CSV file with the date and time embedded in the file name. Unlike Get-AppLockerEvents.ps1, Save-WEFEvents.ps1 takes very few parameters, none of which are mandatory: the directory in which to save the file; the number of days ago from which to start retrieving data, and a name to insert into the file name. *(See sample output in the Samples.zip, SampleOutput-SaveWEFEvents folder.)*

**Generate-EventWorkbook.ps1** takes a CSV file created from Get-AppLockerEvents.ps1 or Save-WEFEvents.ps1 and produces a multi-tabbed Excel workbook containing a variety of summary and detailed information that supports advanced analysis. Data from this workbook can be pasted directly into the customization input scripts to tune the policies. *(See sample output in the Samples.zip, SampleOutput-GenerateEventWorkbook folder.)*

### Using Get-AppLockerEvents.ps1

By default, Get-AppLockerEvents.ps1 retrieves all errors and warnings from both the EXE/DLL and MSI/Script event logs on the local computer. It filters out events involving PowerShell script-policy test files[[16]](#footnote-17), synthesizes additional event data, generates tab-delimited CSV sorted by the content in the order of their columns, and writes it to the pipeline. Its command-line parameters enable you to write to Excel, output PowerShell objects, omit fields from the output, change which event types are reported, and pick different event sources. The following sections describe options in more detail.

Note: with PowerShell v5.x, when AppLocker reports an error or warning for a PowerShell script, it does not mean that the script was not allowed to execute. It means only that it ran in ConstrainedLanguage mode. See the Appendix about PowerShell and whitelisting for more information.

#### Get-AppLockerEvents.ps1 to Excel

Add the -Excel switch to the command line and Get-AppLockerEvents.ps1 writes its data to a formatted Excel spreadsheet instead of to the PowerShell pipeline. The formatting enables sorting and filtering on any field. The “Event data and synthesized data” section below describes the content of these fields.

#### Get-AppLockerEvents.ps1’s CSV output

Save the script’s CSV output to a file if you don’t have Excel installed, or if you want to create a multi-tabbed workbook with Generate-EventWorkbook.ps1. When writing the CSV output to a file, ensure that any Unicode and other special characters in the event data such as ® and ™ survive the pipeline by setting the OutputEncoding preference variable like this:

$OutputEncoding = [System.Text.ASCIIEncoding]::Unicode  
.\Get-AppLockerEvents.ps1 | Out-File -Encoding unicode *$filename*

#### Get-AppLockerEvents.ps1’s PSCustomObject output

Add the -Objects switch to the command line and Get-AppLockerEvents.ps1 outputs objects to the pipeline instead of CSV text. You can then perform all kinds of manipulations with them. For example:

# Retrieve events and convert to objects  
$events = .\Get-AppLockerEvents.ps1 -Objects  
# How many events?  
$events.Count  
# Show generic paths of all unsigned files that were blocked  
$events | where { $\_.PublisherName -eq "-" } | foreach { $\_.GenericPath } | Sort-Object -Unique

Here’s a slightly more elaborate example. It retrieves all events (not just warnings and errors) after 2:09pm on December 15 2018 with “ONEDRIVE” in the file path, sorts them chronologically, and reports the event time, the path, the process ID, and whether it was allowed or blocked (this is one line):

.\Get-AppLockerEvents.ps1 -AllEvents -FromDateTime "12/15/2018 14:09" -Objects | where { $\_.GenericDir.Contains("ONEDRIVE") } | sort EventTime | ft EventTime, GenericPath, PID, EventType -AutoSize

The column names listed in the “Event data and synthesized data” section below are also the property names that these objects expose.

#### Event data and synthesized data

Get-AppLockerEvents.ps1 returns data as reported in the event log sources, and synthesizes data that facilitates the aggregation of information from multiple computers and users. The “OriginalPath” data represents the file path reported in the event log. If the same file is executed from the same subdirectory within, for example, 100 different user profile directories, it will be reported as 100 unique file paths, which makes it harder to see that they are the same. Get-AppLockerEvents.ps1 synthesizes the “GenericPath” field, which replaces user-profile patterns in file paths with an environment variable (%LOCALAPPDATA%, %APPDATA%, or %USERPROFILE%), so that those 100 different paths report the same GenericPath. Get-AppLockerEvents.ps1 also creates the GenericDir field, which is the GenericPath’s parent directory. The GenericDir can be useful when specifying unsafe paths to scan (UnsafePathsToBuildRulesFor.ps1).

By default, Get-AppLockerEvents.ps1 outputs all the fields listed in the following table in the order listed. Command-line switches allow you to omit any of the fields. Rows are sorted alphabetically in column order and any duplicate rows are removed. In other words, with no fields omitted from the output the rows are sorted first by GenericPath, then by GenericDir, then by OriginalPath, etc. With fields such as EventTime, PID (Process ID), MachineName included in the output, duplicates are extremely unlikely. With those fields and user-specific fields such as UserSID, UserName, and OriginalPath excluded, duplicates are much more likely which will reduce the number of rows reported.

|  |  |
| --- | --- |
| **Column name** | **Description** |
| GenericPath | GenericPath is the original file path with "%LOCALAPPDATA%" replacing the beginning of the path name if it matches the typical pattern "C:\Users\*username*\AppData\Local". Makes similar replacements for "%APPDATA%" or "%USERPROFILE%" if LOCALAPPDATA isn't applicable. |
| GenericDir | GenericDir is the directory-name portion of GenericPath (i.e., with the filename removed). |
| OriginalPath | OriginalPath is the file path exactly as reported in the AppLocker event log data. If a file is used by multiple users, OriginalPath often includes differentiating information such as user profile name. Omitting this field can be useful when aggregating data from many users running the same programs. |
| FileName | FileName is the logged filename (including extension) by itself without path information. |
| FileExt | FileExt is the file extension of the logged file. This can be useful to track files with non-standard file extensions. |
| FileType | FileType is "EXE," "DLL," "MSI," or "SCRIPT." |
| PublisherName | For signed files, PublisherName is the distinguished name (DN) of the file's digital signer. PublisherName is blank or just a hyphen if the file is not signed by a trusted publisher. |
| ProductName | For signed files, ProductName is the product name taken from the file's version resource. |
| BinaryName | For signed files, BinaryName is the "OriginalName" field taken from the file's version resource. |
| FileVersion | For signed files, FileVersion is the binary file version taken from the file's version resource. |
| Hash | The Hash field, if included, represents the file's SHA256 hash. In addition to being incorporated in rule data, the hash data can help determine whether two files are identical. |
| UserSID | UserSID is the security identifier (SID) of the user that ran or tried to run the file. Omitting this field can be useful when aggregating data from many users running the same programs. |
| UserName | UserName is the result of SID-to-name translation of the UserSID value performed on the local computer. Omitting this field can be useful when aggregating data from many users running the same programs.  Note: If the SID cannot be translated on the system where you run Get-AppLockerEvents.ps1, this field’s value is “[[[built-in local admin]]]” if the SID ends with -500, or “[[[Not translated]]]” otherwise. |
| MachineName | MachineName is the computer name on which the event was logged. Omitting this field can be useful when aggregating data forwarded from many computers. |
| EventTime | EventTime is the date and time that the event occurred, in the computer's local time zone and rendered in this sortable format "yyyy-MM-ddTHH:mm:ss.fffffff". For example, June 13, 2018, 6:49pm plus 17.7210233 seconds is reported as 2018-06-13T18:49:17.7210233. Omitting this field is useful when you want to get at most one event for every file referenced.  Note that Excel does not recognize this precise representation as a date or a time and treats it only as text. You can sort it alphabetically, which corresponds to a chronological sort. Excel does recognize the less-precise EventTimeXL column, described next, as a date/time. |
| EventTimeXL | EventTimeXL is the date and time that the event occurred, in the computer's local time zone and rendered in a format that Excel recognizes as a date/time, so that a column’s filter dropdown can render a tree view, as shown in the screenshot below. As with EventTime, omitting this field is useful when you want to get at most one event for every file referenced. |
| PID | PID is the process ID. It can be used to correlate EXE files and other file types, including scripts and DLLs. Note that a PID is a unique identifier only on the computer the process is running on and only while it is running. When the process exits, the PID value can be assigned to another process. |
| EventType | EventType is "Information," "Warning," or "Error," which can be particularly helpful with -AllEvents, as it's not otherwise possible to tell whether the file was allowed. |

To omit a field, add a command-line option beginning with -No and the column name. For example, to omit the EventTime field, add **‑NoEventTime** to the command line. Note that Generate-EventWorkbook.ps1 produces a workbook with several summary views so you might not need to perform this kind of filtering.

You can also limit events to those after a specific date and time, before a specific date and time, or within a date-and-time range. The **‑FromDateTime** and **‑ToDateTime** parameters each accept any text that can be converted to a [datetime] object. Examples:

# Retrieve events logged from midnight June 13, 2018 and later  
.\Get-AppLockerEvents.ps1 -FromDateTime "6/13/2018"  
# Retrieve events logged from June 13, 2018, 2pm and later, two different ways  
.\Get-AppLockerEvents.ps1 -FromDateTime "6/13/2018 14:00"  
.\Get-AppLockerEvents.ps1 -FromDateTime "6/13/2018 2pm"  
# Retrieve events between 2pm and 3pm on June 13, 2018  
.\Get-AppLockerEvents.ps1 -FromDateTime "6/13/2018 14:00" -ToDateTime "6/13/2018 3pm"

#### Event types and sources

With no other switches, Get-AppLockerEvents.ps1 retrieves error and warning events from both the EXE/DLL and MSI/Script logs. You can retrieve only warnings with **‑WarningOnly** or only errors with **‑ErrorOnly**. If you want to see the Informational events, use **‑Allowed**. The **‑AllEvents** switch reports all events. You can retrieve events from just one of the logs with **-ExeAndDllOnly** or **‑MsiAndScriptOnly**.

On a Windows Event Collector, use the **‑ForwardedEvents** switch to retrieve AppLocker events from the Forwarded Events log. Finally, you can retrieve AppLocker events from one or more saved .evtx files with **‑EvtxLogFilePaths**.

#### “Noise” events and “filtered machines”

Get-AppLockerEvents.ps1 filters out events from randomly-named PowerShell script-policy test files in users’ temp directories, which are logged every time a PowerShell console starts and are described in the “PowerShell and whitelisting” appendix. If for some reason you want to see these events, you can with the **‑NoPSFilter** switch.

AutoNGEN[[17]](#footnote-18) is designed to optimize the execution of .NET programs and Windows Store (“modern”) apps by pre-compiling regularly used apps into native EXE and DLL files in the user’s profile. It’s not possible to create safe AppLocker rules for these files in advance or for all users, and AppLocker blocks them when the app tries to use them, which can result in a lot of logged errors or warnings. The app will still work, though, by using the original files when it can’t load the optimized ones, so you should just ignore those events. Get-AppLockerEvents.ps1 doesn’t filter them out by default, but you can do so with **‑NoAutoNGEN** on the command line.

When you have configured Windows Event Forwarding, it’s possible for computers that subscribe never to have any events to send, or to send only events that Get-AppLockerEvents.ps1 filters out. At the same time, it’s useful to know that these machines have successfully subscribed. Get-AppLockerEvents.ps1 lists each of these computers in its output with a “pseudo-event” with the MachineName field set the computer name, FileType set to “NONE,” EventType set to “FILTERED,” and all other fields blank. You can omit the listing of these “filtered machines” with **‑NoFilteredMachines**.

### Using Save-WEFEvents.ps1

**Save-WEFEvents.ps1** is a wrapper around Get-AppLockerEvents.ps1 that’s designed specifically for use on a Windows Event Collector server that receives AppLocker events.

Because forwarded events are logged into the server’s Forwarded Events log, Save-WEFEvents retrieves events from that log rather than from the default AppLocker logs. Save-WEFEvents generates a filename with the current date and time embedded into the file name; for example, *ForwardedEvents-20180613-1358.csv*., generated at 1:58pm on June 13, 2018. Save-WEFEvents.ps1 assumes that only warnings and events are forwarded and that informational events aren’t, so it reports only warnings and errors.

Save-WEFEvents.ps1 takes three optional parameters: the **‑rootdir** parameter lets you specify the directory in which to save the CSV file; the **‑daysBack** parameter lets you specify the number of days ago from which to start retrieving events; and the **‑label** parameter lets you specify text that gets included in the file name. Consider this example:

.\Save-WEFEvents.ps1 -rootdir c:\temp -daysBack 7 -label PILOT

This creates a CSV file containing forwarded events from the past seven days to a file called c:\temp\ForwardedEvents-PILOT-20180614-1249.csv. (The script was executed at 12:49pm on June 14, 2018.)

Because it is designed to be run on a Windows Event Collector server where Microsoft Office probably is not installed, Save-WEFEvents.ps1 does not include an option to create an Excel spreadsheet. You can move the CSV file to a workstation that has Excel and run Generate-EventWorkbook.ps1.

### Using Generate-EventWorkbook.ps1

Running Generate-EventWorkbook.ps1 is very simple. It takes one required parameter: the path to a CSV file created from Get-AppLockerEvents.ps1 or Save-WEFEvents.ps1, which supplies the data that goes into the workbook. If you add the **‑SaveWorkbook** switch, it saves the workbook to the same path as the input file but with a .xlsx extension.

Generate-EventWorkbook.ps1 takes the data from the input CSV and “slices and dices” the data, creating a multi-tabbed workbook containing the following tabs:

|  |  |
| --- | --- |
| **Tab name** | **Description** |
| ***Summary*** | Summary information, including name of source file; first and last date/time of observed events; total number of events, signed-file, and unsigned-file events; and number of computers reporting. |
| ***Machines and event counts*** | Sorted list of all computers reporting and the number of events from each. |
| ***Publishers and event counts*** | Sorted list of the publishers of signed files that were reported, and the number of events associated with each publisher. If the publisher name is “-“ it refers to unsigned files, and the count is the number of events involving unsigned files. |
| ***Publisher-product combinations*** | Sorted list of the publisher and product combinations that were reported, with duplicates removed. |
| ***Signed file info*** | Sorted list reporting every combination of publisher, product name, generic path, file name, and file type, with duplicates removed. |
| ***Unsigned file info*** | Sorted list reporting the generic path, file name, file type, and hash of all unsigned files reported, with duplicates removed. |
| ***Users and event counts*** | Sorted list of all users reporting and the number of events from each. |
| ***Files by user*** | Sorted list of all users reporting events, and the GenericPath, PublisherName, and ProductName for each, with duplicates removed. |
| ***Files by user (details)*** | Sorted list of all users reporting events, grouped by MachineName and then sorted chronologically (EventTimeXL), with GenericPath, PublisherName, and ProductName. |
| ***Full details*** | Equivalent to the output of Get-AppLockerEvents, with all captured details. |

### Making adjustments

The information retrieved by the three scripts just described gives you the best visibility you can get into how your policies are working and precisely what changes you might need. Warnings and errors fall into three primary buckets:

* One or more users have a legitimate need to run the file. The event indicates a gap in the rule set that needs to be fixed.
* One or more users are intentionally trying to run something that they shouldn’t. AppLocker is doing its job. Don’t adjust the rules.
* The file is malware. AppLocker is doing its job. Don’t adjust the rules.

Before proceeding, you must determine into which bucket a given file falls. This is often going to be a judgement call, particularly with respect to the first two bullets. Just because AppLocker blocks a file that isn’t malware, does that imply a legitimate user need that organization policy must accommodate?

#### Altering the program

If you decide that users have legitimate need to run a program that is incompatible with the existing rules, the first step should be to determine whether it’s possible to change the programs to conform with recommended practices. Programs should be installed to safe directories, preferably under Program Files or Program Files (x86). If programs must be installed to unsafe directories such as the user profile, they should be digitally signed and include version resource information, as Publisher rules are preferable to Hash rules.

Advise your developers to follow these guidelines:

* Install executable files (for example, EXE, DLL, and scripts) only to ProgramFiles[[18]](#footnote-19) subdirectories, not to ProgramData, user profiles, or custom subdirectories under the drive root;
* Do not relax permissions on any subdirectories or files under the ProgramFiles directories. The default permissions disallow non-administrative users from creating, modifying, or deleting content in those locations.
* If you need to put data in a machine-wide location, create a custom subdirectory under ProgramData, not under ProgramFiles.

Programs that adhere to the above guidelines are *always* compatible with AaronLocker rules and do not need any customizations to work correctly.

If you can’t put the files into ProgramFiles, you will need to apply at least some customizations to the AaronLocker rules. These guidelines will help simplify those customizations:

* If the files are in an unsafe directory that isn’t in the user profile – such as a custom “C:\Apps” under the drive root – see whether the program will still work if permissions are locked down to make it a safe directory in which no non-administrators are granted more than Read + Execute.
* If the files must remain in an unsafe directory, advise your developers to sign all files and to incorporate version resource information including a product name in all binary files. For example, ClickOnce apps always install to the user profile under %LOCALAPPDATA%\Apps\2.0 and therefore always require some customization of AaronLocker rules to run. Note that signing files in ClickOnce apps must be performed after the files are built but prior to packaging, so it must be part of the developer process. Without these steps, the only way to allow these programs to run is with hash rules that must be updated each time the app is updated.

#### Customizing the rules

If, after altering the program as much as possible according to the previous section, you still need to allow file execution from locations other than safe ProgramFiles subdirectories, this section describes how.

**Rules from scanning directories and files**

If the files are in a safe directory, add the directory to GetSafePathsToAllow.ps1. You can get the directory information from several of the tabs in the workbook produced by Generate-EventWorkbook.ps1. However, the event data cannot indicate whether the path is safe. You can run Scan-Directories.ps1 on an end user system to determine directory safety.

If the files are in an unsafe directory and you have the files installed on your rule-building system, create a new entry in UnsafePathsToBuildRulesFor.ps1 that references the path or paths, and run Create-Policies.ps1. If instead you have the files on another system that you can scan, copy the AaronLocker scripts onto that system, create a new UnsafePathsToBuildRulesFor.ps1 entry, run Create-Policies.ps1, then copy the generated files from MergeRules-Dynamic on the scanned system to MergeRules-Static on your rule-building system. (You can also instead use BuildRulesForFilesInWritableDirectories.ps1 in the Support directory. That’s the script that performs the bulk of the work for each of the unsafe-paths entries. Copy the resulting file to MergeRules-Static on your rule-building system.)

**Rules from event data only**

If you don’t have access to the files that triggered the events, you can still customize rules using event data reported by the event-collecting scripts or the workbook generated by Generate-EventWorkbook.ps1. Note that if all you have is event data, you cannot determine conclusively whether the files are in safe or unsafe directories, so you should assume the directories are unsafe.

If the files are signed, add one or more entries to TrustedSigners.ps1 with the publisher name and optionally with more detail, depending on how restrictive you want to make the rules. If you want to allow everything from the publisher, use just the PublisherName property, which you can find on many of the workbook tabs. You can also restrict only to a specific product by that publisher, specific files within that product, and even block versions below a particular version number. The event fields you can use with TrustedSigners.ps1 are PublisherName, ProductName, BinaryName, FileVersion, and FileType. For more information, see the TrustedSigners.ps1 section in Appendix C.

If the files are not signed, add one or more entries in HashRuleData.ps1. The event fields you need are FileName, FileType, and Hash. You can find these in the workbook on the “Unsigned file info” or “Full details” tabs. For more information, see the HashRuleData.ps1 section in Appendix C.

If you can, it’s preferable to scan installed files rather than to rely solely on event data. Event data tells only what file execution was attempted from a directory. It doesn’t report other files associated with the app that might need to be executed in the future.

## Whitelisting beyond AppLocker

AppLocker is a powerful whitelisting tool when configured well, but even with very strict and tightly-scoped rules, it still leaves gaps that enable non-administrators to run unapproved code that can cause harm. This section describes some of those gaps and some techniques to close them.

### Microsoft Office macros

Microsoft Office applications come with Visual Basic for Applications (VBA), which supports automation features such as macros. VBA is essentially a full-featured programming platform that is similar in capability to the old Visual Basic 6.0 programming tool, including the ability to invoke Win32 APIs, start COM and DCOM components, and interoperate with .NET code. It adds full integration with Office apps’ object models and the ability to embed code directly into documents. Macros’ flexibility, power, and portability bring tremendous benefit to users and organizations, but those same attributes can be used for ill: Office macros are a common delivery technique for malware, and the most ubiquitous and well-known whitelisting bypass mechanism available.

If Office apps are installed and allowed to run, AppLocker rules can do very little to restrict macro execution. Commands in macros are acted upon within Office processes without running files in the user profile or creating new files that AppLocker would block. If a macro downloads or creates a new file in a user-writable directory, AppLocker *can* prevent that file from running, but that’s about it. This section describes other means to restrict potentially unsafe code execution from Office VBA.

VBA code execution can be defined and executed from multiple places including:

* Embedded in a Word, Excel, or PowerPoint document.
* User-defined macros stored in the user’s profile.
* The VBA editor’s “Immediate window”.

#### Macros embedded in a Word, Excel, or PowerPoint document

This is where “macro viruses” and “macro malware” come from: a document containing malicious macros gets emailed to or downloaded by victims, and the macro executes. Macro viruses used to be more prevalent than they are now because it used to be possible to configure a macro to run automatically when the document was opened. Embedded macros are now disabled by default and won’t run without some user interaction. Some users can still be tricked into running them, though.

Office’s legacy binary file formats such as .doc, .xls, and .ppt can always contain embedded macros. The Office Open XML (OOXML) document formats introduced in Office 2007 (e.g., .docx, .xlsx, and .pptx) cannot; only the macro-enabled versions of those documents (.docm, .xlsm, and .pptm) can contain embedded macros.[[19]](#footnote-20)

The rest of this section describes several ways you can defend against the execution of untrusted macros embedded in Office documents.

**Apply Microsoft’s Office 2016 security baseline**, part of the Security Compliance Toolkit (SCT).[[20]](#footnote-21) With the baseline applied, embedded macros cannot be enabled unless they are digitally signed by a trusted publisher. (Note, however, that Office allows non-administrative users to choose which publishers they want to trust.) The baseline also disables macros embedded in Office documents that came from the internet, including through email from an external sender, even if the macros are signed.[[21]](#footnote-22)

Note: While not related to macros, the Office 2016 baseline also introduces a custom setting that blocks a common exploit technique that leads to arbitrary code execution. Vulnerabilities in Adobe Flash are often exploited by sending the victim an Office document that contains malformed data with a reference that activates Flash and triggers the exploit code. The Office 2016 baseline prevents the Flash ActiveX control from being loaded by Office applications, blocking the exploit.[[22]](#footnote-23)

**Configure Attack Surface Reduction (ASR) in Windows Defender Exploit Guard (WDEG).** On Windows 10 version 1709 and newer, Attack Surface Reduction offers several settings designed specifically to block macro-based malware and other Office exploits, including:

* Block Office applications from creating child processes
* Block Office applications from creating executable content
* Block Office applications from injecting code into other processes
* Block Win32 API calls from Office macro

For more information about WDEG and ASR, see <https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-exploit-guard/attack-surface-reduction-exploit-guard>.

**Block dangerous file types.** A couple of additional steps you can consider are to configure file-block settings in Group Policy to prevent Office apps from loading legacy file formats, and to configure your mail attachment filters to disallow macro-enabled documents sent from outside your organization.

#### User-defined macros stored in the user’s profile

Users can record and edit macros that are always available to them and not just from within specific documents. Rather than embedded in documents, these macros are stored in an app-specific file in the user’s profile. A user’s Word macros are stored in %APPDATA%\Microsoft\Templates\Normal.dotm, and Excel macros in %APPDATA%\Microsoft\Excel\xlstart\Personal.xlsb.

A user’s private macro store is generally not an easy or useful target for malicious external actors, but it does afford the interactive user an opportunity to define and run arbitrary code. The Office 2016 baseline’s macro-signing requirement also applies to user-defined macros, which raises the bar if users don’t know how to sign their macro code. However, Office allows users to apply and trust a self-signed certificate to sign macro code for their own use, and even provides a graphical tool to create a self-signed certificate. If interactive users creating and running arbitrary code in their own macros is a threat you are concerned with, there is little you can do other than to disable VBA across all of Office through Group Policy. If that’s not feasible, it might be somewhat worthwhile to blacklist Office’s selfcert.exe.

#### The VBA editor’s “Immediate window”

The VBA editor’s Immediate window enables the interactive user to run VBA commands one line at a time. Among other things, it is helpful for debugging macros, trying out lines of code one at a time, and for querying and modifying attributes of currently-open documents. The user can instantiate ActiveX objects and invoke their methods. In other words, pretty much anything you can do from a macro you can also do from the Immediate window. Note that in terms of risk, the Immediate window is of use as a whitelist-bypass mechanism only to an interactive user.

Policy restrictions on unsigned macros are only partially effective on the Immediate window. If the current VBA project contains unsigned macros, then the user can’t run anything in the Immediate window. However, if the current project has no macros or has only signed macros, there is no constraint on what the user can run in the Immediate window.

The only way to block access to the Immediate window is to disable VBA across all of Office through Group Policy.

### Oracle Java and other programming platforms

Oracle Java and other programming/scripting platforms present similar challenges to those posed by Office macros. If the tool is installed to Program Files, it’s allowed to run. These products read files (e.g., .jar files) that are simply data files as far as AppLocker is concerned and therefore aren’t evaluated or blocked. The platform program then interprets the contents of these files and executes arbitrary actions accordingly. If these tools are installed, they can be easy whitelist bypasses. In addition, until recently Java was the most widely attacked software product in the world. That has changed in part because several major browsers dropped support for it. But older versions with security vulnerabilities can still lead to exploitation and the running of arbitrary code within an approved program.

Recommended mitigations:

* Don’t install Java or other programming/scripting platforms unless they are absolutely needed.
* Keep them up to date – do not retain older versions that have known security vulnerabilities.
* See whether the tool offers its own control mechanisms to restrict execution only to pre-approved content.

### Unusual EXE/DLL combinations

Security researchers occasionally find interesting ways to bypass whitelisting using Windows’ own tools. Two of them involve a Windows EXE loading a specific Windows DLL and then hooking in arbitrary content and executing it. The command lines are complex and I can’t imagine even an extremely gullible user being tricked into executing them. I don’t consider them very high risk, but let’s try to stop them anyway.

In one case, RunDll32.exe loads mshtml.dll; in the other, Regsvr32.exe loads scrobj.dll. In neither case is it possible to block either the EXE or the DLL outright, as the files are needed for Windows’ normal operations. At the same time, there never is a need for a user to load mshtml.dll into RunDll32.exe or scrobj.dll into Regsvr32.exe. On Windows 10, we can use code integrity rules to disallow those specific combinations.

*[[[ Working on this; not ready to release yet. ]]]*

# Appendix A: Main script reference

This appendix describes detailed usage for the seven scripts in the AaronLocker root directory. You can get the same information from within PowerShell using Get-Help; for example:

Get-Help .\Get-AppLockerEvents.ps1 -Detailed

## Create-Policies.ps1

Builds comprehensive and robust AppLocker "audit" and "enforce" rules to mitigate against users running unauthorized software, customizable through simple text files. Writes results to the Outputs subdirectory.

***Syntax***

Create-Policies.ps1 [-Rescan] [[-ForUser] <String>] [-Excel] [<CommonParameters>]

***Parameters***

**-Rescan [<SwitchParameter>]**

If this switch is set, this script scans the Windows and ProgramFiles directories for user-writable subdirectories, and captures data about EXE files to blacklist.

If the results from a previous scan are found in the expected location and this switch is not specified, the script does not perform those scans. If those results are not found, the script performs the scan even if this switch is not set.

It is STRONGLY recommended that the scanning be performed with administrative rights.

**-ForUser <String>**

If scanning a system with an administrative account with a need to inspect another user's profile for "unsafe paths," specify that username with this optional parameter. E.g., if logged on and scanning with administrative account "abby-adm" but need to inspect $env:USERPROFILE belonging to "toby", use -ForUser toby.

**-Excel [<SwitchParameter>]**

If specified, also creates Excel spreadsheets representing the generated rules.

***Description***

Create-Policies.ps1 generates comprehensive "audit" and "enforce" AppLocker rules to restrict non-admin code execution to "authorized" software, in a way to minimize the need to update the rules. Broadly speaking, "authorized" means that an administrator put it on the computer *or* created a rule specifically for that item.

Supported operating systems include Windows 7 and newer, and Windows Server 2008 R2 and newer.

Rules cover EXE, DLL, Script, and MSI; on Windows 8.1 and newer, rules also cover Packaged apps.

Allows non-admin execution from the Windows and ProgramFiles directories, EXCEPT:

* Identifies user-writable subdirectories and disallows execution from those directories;
* Disallows execution of programs that run user-supplied code (e.g., mshta.exe);
* Disallows execution of programs that non-admins rarely need but that malware/ransomware authors are known to use (e.g., cipher.exe);

Allows execution from identified "safe" paths (non-admins cannot write to them);

Allows execution of specifically authorized code in user-writable ("unsafe") directories.

Rule implementation:

AppLocker rule types include path rules, publisher rules, and hash rules.

Rules allowing execution from "safe" locations are implemented using path rules.

User-writable subdirectories of the Windows and ProgramFiles directories are identified using Sysinternals AccessChk.exe. Exceptions for those subdirectories are implemented within path rules.

Exceptions for "dangerous" programs (e.g., mshta.exe, cipher.exe) are generally implemented with publisher rules.

Rules allowing execution of EXE, DLL, and script files from user-writable directories are implemented with publisher rules when possible, and hash rules otherwise, with options for the granularity of Publisher rules.

Publisher rules can also be created allowing execution of anything signed by a particular publisher, or a specific product by a particular publisher.

Scanning for user-writable subdirectories of the Windows and ProgramFiles directories can be time-consuming. The script writes results to text files in an intermediate subdirectory. The script runs the scan if those files are not found OR if the -Rescan switch is specified.

It is STRONGLY recommended that the scanning be performed with administrative rights.

Once scans have been performed, scanned output can be copied to another machine and rules can be maintained without needing to rescan.

Dependencies:

* PowerShell v5.1 or higher (Windows Management Framework 5.1 or higher)
* Current (or recent) version of Sysinternals AccessChk.exe, either in the Path or in the same directory as this script.
* Scripts and support files included in this solution (some are in specific subdirectories).

Sysinternals AccessChk is available here:

<https://technet.microsoft.com/sysinternals/accesschk>

<https://download.sysinternals.com/files/AccessChk.zip>

<https://live.sysinternals.com/accesschk.exe>

or run Support\DownloadAccesschk.ps1, which downloads AccessChk.exe to the main AaronLocker directory.

## Scan-Directories.ps1

Scan directories to identify files that might need additional AppLocker rules.

***Syntax***

Scan-Directories.ps1 [-WritableWindir] [-WritablePF] [-SearchProgramData] [-SearchOneUserProfile] [-SearchAllUserProfiles] [-SearchNonDefaultRootDirs] [-DirsToSearch <String[]>] [-NoPEFiles] [-NoScripts] [-NoMSIs] [-JS] [-DirectoryNamesOnly] [-Excel] [<CommonParameters>]

Scan-Directories.ps1 [-FindNonDefaultRootDirs] [<CommonParameters>]

***Parameters***

**-WritableWindir [<SwitchParameter>]**

If this switch is specified, searches user-writable subdirectories under %windir% according to results of the last scan performed by Create-Policies.ps1.

**-WritablePF [<SwitchParameter>]**

If this switch is specified, searches user-writable subdirectories under the %ProgramFiles% directories according to results of the last scan performed by Create-Policies.ps1.

**-SearchProgramData [<SwitchParameter>]**

If this switch is specified, searches the %ProgramData% directory hierarchy, which can contain a mix of "safe" and "unsafe" directories.

**-SearchOneUserProfile [<SwitchParameter>]**

If this switch is specified, searches the user's profile directory.

**-SearchAllUserProfiles [<SwitchParameter>]**

If this switch is specified, searches from the root directory of all users' profiles (C:\Users)

**-SearchNonDefaultRootDirs [<SwitchParameter>]**

If this switch is specified, search all non-standard directories in the %SystemDrive% root directory. These directories often contain LOB applications.

**-DirsToSearch <String[]>**

Specifies one or more directories to search.

**-NoPEFiles [<SwitchParameter>]**

If this switch is specified, does not search for Portable Executable files (EXE/DLL files)

**-NoScripts [<SwitchParameter>]**

If this switch is specified, does not search for script files.

**-NoMSIs [<SwitchParameter>]**

If this switch is specified, does not search for MSI files.

**-JS [<SwitchParameter>]**

If this switch is specified, report .js files as script files; otherwise, skip .js files entirely.

**-DirectoryNamesOnly [<SwitchParameter>]**

If this switch is specified, reports the names and "safety" of directories that contain files of interest but no file information.

**-Excel [<SwitchParameter>]**

If this switch is specified, outputs to formatted Excel worksheet instead of to pipeline

**-FindNonDefaultRootDirs [<SwitchParameter>]**

If this switch is specified, identifies non-standard directories in the %SystemDrive% root directory. These directories often contain LOB applications.

This switch cannot be used with any other options.

***Description***

Produces tab-delimited CSV or an Excel worksheet listing files in various directories that might need additional AppLocker rules to allow them to execute. Optionally, the script can list non-standard directories in the %SystemDrive% root directory. These directories might require additional scanning.

The script searches specified directory hierarchies for MSIs and scripts (according to file extension), and EXE/DLL files regardless of extension. That is, a file can be identified as a Portable Executable (PE) file (typically an EXE or DLL) even if it has a non-standard extension or no extension.

Output columns include:

* IsSafeDir - indicates whether the file's parent directory is "safe" (not user-writable) or "unsafe" (user-writable);
* File type - EXE, DLL, MSI, or Script;
* File extension - the file's extension;
* File name - the file name without path information;
* File path - Full path to the file;
* Parent directory - The file's parent directory;
* Publisher name, Product name, Binary name, Version - signature and version resource information that can be used in publisher rules;
* Hash - the file’s hash;
* CreationTime, LastAccessTime, LastWriteTime - the file's timestamps according to the file system;
* File size.

Directories that can be searched:

* WritableWindir - writable subdirectories of the %windir% directory, based on results of the last scan performed by Create-Policies.ps1;
* WritablePF - writable subdirectories of the %ProgramFiles% directories, based on results of the last scan performed by Create-Policies.ps1;
* SearchProgramData - the %ProgramData% directory hierarchy;
* SearchOneUserProfile - the current user's profile directory;
* SearchAllUserProfiles - the root directory of user profiles (C:\Users);
* SearchNonDefaultRootDirs - all non-default directories in the %SystemDrive% root directory.
* DirsToSearch - one or more caller-specified, comma-separated directory paths.

Results can be imported into Microsoft Excel and analyzed.

Note that results from this script do not necessarily require that rules be created: this is just an indicator about files that \*might\* need rules, if the files need to be allowed.

***Examples***

PS C:\>Scan-Directories.ps1 -SearchOneUserProfile -DirsToSearch H:\

Searches the user's profile directory and the H: drive.

## ExportPolicy-ToExcel.ps1

Turns AppLocker policy into a more human-readable Excel worksheet.

***Syntax***

ExportPolicy-ToExcel.ps1 [-Local] [<CommonParameters>]

ExportPolicy-ToExcel.ps1 [-AppLockerXML <String>] [-SaveWorkbook] [<CommonParameters>]

ExportPolicy-ToExcel.ps1 [-AppLockerCSV <String>] [-SaveWorkbook] [<CommonParameters>]

***Parameters***

**-Local [<SwitchParameter>]**

If this switch is specified, the script processes the computer's local AppLocker policy. If no parameters are specified or this switch is set to -Local:$false, the script processes the computer's effective AppLocker policy.

**-AppLockerXML <String>**

If this parameter is specified, AppLocker policy is read from the specified exported XML policy file.

**-AppLockerCSV <String>**

If this parameter is specified, AppLocker policy is read from the specified CSV file previously created from ExportPolicy-ToCsv.ps1 output.

**-SaveWorkbook [<SwitchParameter>]**

If set, saves workbook to same directory as input file with same file name and default Excel file extension.

***Description***

The script gets AppLocker policy from one of four sources, imports it into a new Excel instance, and formats it.

The four source options are:

* Current effective policy (default behavior -- use no parameters);
* Current local policy (use -Local switch);
* Exported AppLocker policy in an XML file (use -AppLockerXML parameter with file path);
* Output previously captured from ExportPolicy-ToCsv.ps1 (use -AppLockerCSV with file path);

This script depends on ExportPolicy-ToCsv.ps1, which should be in the Support subdirectory.

It also depends on Microsoft Excel's being installed.

The three command line options (-Local, -AppLockerXML, -AppLockerCSV) are mutually exclusive: only one can be used at a time.

***Examples***

-------------------------- EXAMPLE 1 --------------------------

PS C:\>.\ExportPolicy-ToExcel.ps1

Generates an Excel worksheet representing the computer's effective AppLocker policy.

-------------------------- EXAMPLE 2 --------------------------

PS C:\>.\Support\ExportPolicy-ToCsv.ps1 | Out-File .\AppLocker.csv; .\ExportPolicy-ToExcel.ps1 -AppLockerCSV .\AppLocker.csv

Generates an Excel worksheet representing AppLocker policy previously generated from ExportPolicy-ToCsv.ps1 output.

-------------------------- EXAMPLE 3 --------------------------

PS C:\>Get-AppLockerPolicy -Local -Xml | Out-File .\AppLocker.xml; .\ExportPolicy-ToExcel.ps1 -AppLockerXML .\AppLocker.xml

Generates an Excel worksheet representing AppLocker policy exported from a system into an XML file.

## Compare-Policies.ps1

Compares two AppLocker policies.

***Syntax***

Compare-Policies.ps1 [-ReferencePolicyXML] <String> [-ComparisonPolicyXML] <String> [-DifferencesOnly] [-Excel] [<CommonParameters>]

***Parameters***

**-ReferencePolicyXML <String>**

Path to AppLocker policy XML file.

Use "local" to inspect local policy.

Use "effective" to inspect effective policy.

**-ComparisonPolicyXML <String>**

Path to AppLocker policy XML file.

Use "local" to inspect local policy.

Use "effective" to inspect effective policy.

**-DifferencesOnly [<SwitchParameter>]**

If this optional switch is specified, entries that are in both sets and are identical are not reported.

**-Excel [<SwitchParameter>]**

If this optional switch is specified, outputs to a formatted Excel rather than tab-delimited CSV text to the pipeline. Note that when the -Excel switch is not used, line breaks within the CSV text fields are represented as "^|^".

***Description***

Reads two AppLocker policy XML files, canonicalizes and compares the rule information and reports results as tab-delimited CSV, or optionally to an Excel workbook formatted for sorting and filtering.

Output columns are Compare, Rule, Reference, and Comparison.

The "Compare" column is one of the following values:

"==" if values are the same in both rule sets

"<->" if values are present in both rule sets but different

"<--" if the rule exists only in the reference rule set

"-->" if the rule exists only in the comparison rule set

The "Rule" column is either the name of a rule collection (Exe, Dll, Script, etc.) or information about a specific rule.

The "Reference" column shows data from the ReferencePolicyXML parameter.

The "Comparison" column shows data from the ComparisonPolicyXML parameter.

Where the "Rule" column contains just the name of a rule collection, the Reference and Comparison columns indicate whether rules for that collection are "AuditOnly" or "Enabled" (enforced).

Otherwise, the "Rule" column shows information about a specific rule, including: the file type (e.g., Dll, Exe); rule type (Publisher, Path, Hash); Allow or Deny; user/group SID; and rule-type-specific information.

For Publisher rules, the rule-specific information catenates the publisher, product, and binary name. (Product or binary name might be empty.)

For Path rules, the path is the rule-specific information.

For Hash rules, the source file name is the rule-specific information.

The Reference and Comparison columns show more detailed rule-type-specific information about the rule from the Reference and Comparison rule sets:

For Publisher rules: the low and high version numbers that the rule applies to. If the Publisher rule includes exceptions, the raw XML is appended.

For Path rules: exceptions to the rule, sorted.

For Hash rules: the hash algorithm and value.

When a rule set contains overlapping rules (e.g., two separate hashes allowed for the same file name), the detailed information is appended into the Reference or Comparison column.

Note that when the -Excel switch is not used, line breaks within the CSV text fields are represented as "^|^".

***Examples***

.\Compare-Policies.ps1 local effective -DifferencesOnly

Compare local policy against effective policy and report only the differences.

## Get-AppLockerEvents.ps1

Retrieves and sorts event data from AppLocker logs, removes duplicates, and reports as tab-delimited CSV output, PSCustomObjects, or as an Excel worksheet.

***Syntax***

Get-AppLockerEvents.ps1 [-ComputerName <String>] [-ExeAndDllOnly] [-MsiAndScriptOnly] [-ForwardedEvents] [-WarningOnly] [-ErrorOnly] [-Allowed] [-AllEvents] [-FromDateTime <DateTime>] [-ToDateTime <DateTime>] [-NoGenericPath] [-NoGenericDir] [-NoOriginalPath] [-NoFileName] [-NoFileExt] [-NoFileType] [-NoPublisherName] [-NoProductName] [-NoBinaryName] [-NoFileVersion] [-NoHash] [-NoUserSID] [-NoUserName] [-NoMachineName] [-NoEventTime] [-NoEventTimeXL] [-NoPID] [-NoEventType] [-NoAutoNGEN] [-NoPSFilter] [-NoFilteredMachines] [-Excel] [-Objects] [<CommonParameters>]

Get-AppLockerEvents.ps1 [-EvtxLogFilePaths <String[]>] [-WarningOnly] [-ErrorOnly] [-Allowed] [-AllEvents] [-FromDateTime <DateTime>] [-ToDateTime <DateTime>] [-NoGenericPath] [-NoGenericDir] [-NoOriginalPath] [-NoFileName] [-NoFileType] [-NoPublisherName] [-NoProductName] [-NoBinaryName] [-NoFileVersion] [-NoHash] [-NoUserSID] [-NoUserName] [-NoMachineName] [-NoEventTime] [-NoEventTimeXL] [-NoPID] [-NoEventType] [-NoAutoNGEN] [-NoPSFilter] [-NoFilteredMachines] [-Excel] [-Objects] [<CommonParameters>]

***Parameters***

**-ComputerName <String>**

Inspects events on the named remote computer instead of the local computer. Caller must have administrative rights on the remote computer.

**-ExeAndDllOnly [<SwitchParameter>]**

Retrieves only from the EXE and DLL log (doesn't retrieve from the MSI and Script log).

If neither -ExeAndDllOnly or -MsiAndScriptOnly are specified, retrieves from both logs.

**-MsiAndScriptOnly [<SwitchParameter>]**

Retrieves only from the MSI and Script log (doesn't retrieve from the EXE and DLL log).

If neither -ExeAndDllOnly or -MsiAndScriptOnly are specified, retrieves from both logs.

**-ForwardedEvents [<SwitchParameter>]**

Retrieves from the ForwardedEvents log instead of from the EXE/DLL and MSI/Script logs.

**-EvtxLogFilePaths <String[]>**

Specifies path to one or more saved event log files. (Cannot be used with -ComputerName, -ExeAndDllOnly, or -MsiAndScriptOnly.)

**-WarningOnly [<SwitchParameter>]**

Reports only Warning events (AuditOnly mode; "would have been blocked"), instead of Errors + Warnings.

**-ErrorOnly [<SwitchParameter>]**

Reports only Error events (Enforce mode; files actually blocked), instead of Errors + Warnings.

**-Allowed [<SwitchParameter>]**

Reports only Information events (files allowed to run) instead of Errors + Warnings.

**-AllEvents [<SwitchParameter>]**

Reports all Information, Warning, and Error events.

**-FromDateTime <DateTime>**

Reports only events on or after the specified date or date-time. E.g., -FromDateTime "9/7/2017" or -FromDateTime "9/7/2017 12:00:00"

Can be used with -ToDateTime to specify a date/time range. Date/time specified in local time zone.

**-ToDateTime <DateTime>**

Reports only events on or before the specified date or date-time. E.g., -ToDateTime "9/7/2017" or -ToDateTime "9/7/2017 12:00:00"

Can be used with -FromDateTime to specify a date/time range. Date/time specified in local time zone.

**-NoGenericPath [<SwitchParameter>]**

GenericPath is the original file path with "%LOCALAPPDATA%" replacing the beginning of the path name if it matches the typical pattern "C:\Users\[username]\AppData\Local".

Makes similar replacements for "%APPDATA%" or "%USERPROFILE%" if LOCALAPPDATA isn't applicable.

If -NoGenericPath is specified, GenericPath data is not included in the output.

**-NoGenericDir [<SwitchParameter>]**

GenericDir is the directory-name portion of GenericPath (i.e., with the filename removed).

If -NoGenericDir is specified, GenericDir data is not included in the output.

**-NoOriginalPath [<SwitchParameter>]**

OriginalPath is the file path exactly as reported in the AppLocker event log data.

If a file is used by multiple users, OriginalPath often includes differentiating information such as user profile name.

If -NoOriginalPath is specified, OriginalPath data is not included in the output. This can be useful when aggregating data from many users running the same programs.

**-NoFileName [<SwitchParameter>]**

FileName is the logged filename (including extension) by itself without path information.

If -NoFileName is specified, FileName data is not included in the output.

**-NoFileExt [<SwitchParameter>]**

FileExt is the file extension of the logged file. This can be useful to track files with non-standard file extensions.

If -NoFileExt is specified, FileExt data is not included in the output.

**-NoFileType [<SwitchParameter>]**

FileType is "EXE," "DLL," "MSI," or "SCRIPT."

If -NoFileType is specified, FileType data is not included in the output.

**-NoPublisherName [<SwitchParameter>]**

For signed files, PublisherName is the distinguished name (DN) of the file's digital signer. PublisherName is blank or just a hyphen if the file is not signed by a trusted publisher.

If -NoPublisherName is specified, PublisherName data is not included in the output.

**-NoProductName [<SwitchParameter>]**

For signed files, ProductName is the product name taken from the file's version resource.

If -NoProductName is specified, ProductName data is not included in the output.

**-NoBinaryName [<SwitchParameter>]**

For signed files, BinaryName is the "OriginalName" field taken from the file's version resource.

If -NoBinaryName is specified, BinaryName data is not included in the output.

**-NoFileVersion [<SwitchParameter>]**

For signed files, FileVersion is the binary file version taken from the file's version resource.

If -NoFileVersion is specified, FileVersion data is not included in the output.

**-NoHash [<SwitchParameter>]**

The Hash field, if included, represents the file's SHA256 hash. In addition to being incorporated in rule data, the hash data can help determine whether two files are identical.

If -NoHash is specified, the file's SHA256 hash data is not included in the output.

**-NoUserSID [<SwitchParameter>]**

UserSID is the security identifier (SID) of the user that ran or tried to run the file.

If -NoUserSID is specified, UserSID data is not included in the output.

If a file is used by different users, UserSID is differentiating. -NoUserSID can be useful when aggregating data from many users running the same programs.

**-NoUserName [<SwitchParameter>]**

UserName is the result of SID-to-name translation of the UserSID value performed on the local computer.

If -NoUserName is specified, SID-to-name translation is not attempted and UserName data is not included in the output.

If a file is used by different users, UserName is differentiating. -NoUserName can be useful when aggregating data from many users running the same programs.

**-NoMachineName [<SwitchParameter>]**

MachineName is the computer name on which the event was logged.

If -NoMachineName is specified, MachineName data is not included in output. This can be useful when aggregating data forwarded from many computers.

**-NoEventTime [<SwitchParameter>]**

EventTime is the date and time that the event occurred, in the computer's local time zone and rendered in this sortable format "yyyy-MM-ddTHH:mm:ss.fffffff".

For example, June 13, 2018, 6:49pm plus 17.7210233 seconds is reported as 2018-06-13T18:49:17.7210233.

If -NoEventTime is specified, EventTime data is not included in the output. This is useful when you want to get at most one event for every file referenced.

**-NoEventTimeXL [<SwitchParameter>]**

EventTimeXL is the date and time that the event occurred, in the computer's local time zone and rendered in a format that Excel recognizes as a date/time, and its filter dropdown renders in a tree view.

If -NoEventTimeXL is specified, EventTimeXL data is not included in the output. This is useful when you want to get at most one event for every file referenced.

**-NoPID [<SwitchParameter>]**

PID is the process ID. It can be used to correlate EXE files and other file types, including scripts and DLLs.

If -NoPID is specified, the PID is not included in the output.

Note that a PID is a unique identifier only on the computer the process is running on and only while it is running. When the process exits, the PID value can be assigned to another process.

**-NoEventType [<SwitchParameter>]**

EventType is "Information," "Warning," or "Error," which can be particularly helpful with -AllEvents, as it's not otherwise possible to tell whether the file was allowed.

If -NoEventType is specified, EventType data is not included in the output.

**-NoAutoNGEN [<SwitchParameter>]**

If specified, does not report modern-app AutoNGEN files that are unsigned and in the user's profile.

**-NoPSFilter [<SwitchParameter>]**

If specified, does not try to filter out random-named PowerShell scripts used to determine whether whitelisting is in effect.

**-NoFilteredMachines [<SwitchParameter>]**

By default, this script outputs a single artificial "empty" event line for every machine for which all observed events were filtered out.

If -NoFilteredMachines is specified, these event lines are not output.

**-Excel [<SwitchParameter>]**

If this optional switch is specified, outputs to a formatted Excel rather than tab-delimited CSV text to the pipeline.

**-Objects [<SwitchParameter>]**

If this optional switch is specified, outputs PSCustomObjects rather than tab-delimited CSV. (Passes CSV through ConvertFrom-Csv.) This switch is ignored if -Excel is also specified..

***Description***

Any fields can be omitted from the output; removing fields with unique data such as event time can result in removal of more lines that otherwise contain duplicated data. AppLocker logs can be saved event log files, or live event logs on the local or a named remote computer.

Output can be tab-delimited CSV, an array of PSCustomObject, or a formatted Excel worksheet.

By default, retrieves error and warning events from both the EXE/DLL and MSI/Script event logs on the local computer. Live-log options include reading events from a remote computer, reading from one of the EXE/DLL and MSI/Script logs instead of both, or reading from the "Forwarded Events" event log on the local or a remote computer. Optionally, read from one or more saved .evtx files.

By default, retrieves error and warning events. AppLocker in audit mode produces warning events ("would have been blocked"), while enforce mode produces error events ("was blocked"). Optionally, read just errors, just warnings, just information events (file was allowed), or all events.

Data from each event (minus any omitted fields) is turned into a line of tab-delimited CSV. These lines are then sorted and duplicates are removed. When fields containing more unique data are omitted, the remaining data will tend to have more duplication and more lines will be removed. See the detailed parameter descriptions for more information. Random-named temporary files created by PowerShell to test whitelisting policy are filtered out by default.

Use the -ComputerName parameter to name a remote computer from which to retrieve events.

Use the -WarningOnly, -ErrorOnly, -Allowed, or -AllEvents switches to retrieve events other than errors+warnings.

Use the -ExeAndDllOnly or -MsiAndScriptOnly switches to retrieve events only from one of the two live event logs.

Use the -ForwardedEvents switch to read from the ForwardedEvents log instead of from the EXE/DLL and MSI/Script logs.

Use the -EvtxLogFilePaths parameter to name one or more saved event log files to read.

Use the -FromDateTime and -ToDateTime parameters to restrict the date/time range to report.

Use the -NoPsFilter switch not to filter out random-named PowerShell policy test script files.

Use the other -No\* switches to omit fields from the output. -NoEventTime, -NoEventTimeXL, and -NoPID are the most important for reducing output size.

See the detailed parameter descriptions for more information.

***Examples***

-------------------------- EXAMPLE 1 --------------------------

PS C:\>.\Get-AppLockerEvents.ps1 -EvtxLogFilePaths .\ForwardedEvents1.evtx, .\ForwardedEvents2.evtx -NoMachineName -NoEventTime -NoEventTimeXL

Get warning and error events from events exported into ForwardedEvents1.evtx and ForwardedEvents2.evtx; don't include MachineName or EventTime data in the output.

-------------------------- EXAMPLE 2 --------------------------

PS C:\>.\Get-AppLockerEvents.ps1 -NoOriginalPath -NoEventTime -NoEventTimeXL -NoUserSID | clip.exe

Get warning and error events from the EXE/DLL and MSI/Script logs on the local computer, removing user-specific and time-specific fields, with the goal that each referenced file appears at most once in the output, no matter how many users referenced it or how often. Write the output to the Windows clipboard so that it can be pasted into Microsoft Excel.

-------------------------- EXAMPLE 3 --------------------------

PS C:\>.\Get-AppLockerEvents.ps1 -Objects | Where-Object { [datetime]($\_.EventTime) -gt "8/20/2017" }

Get warning and error events from the EXE/DLL and MSI/Script logs on the local computer since August 20, 2017. It converts output into objects, and pipes those objects into a filter that passes only events with event dates after midnight, August 20, 2017.

-------------------------- EXAMPLE 4 --------------------------

PS C:\>.\Get-AppLockerEvents.ps1 -FromDateTime "8/1/2017" -ToDateTime "9/1/2017"

Gets warning and error events from the EXE/DLL and MSI/Script logs on the local computer between Aug 1, 2017 00:00:00 and Sept 1, 2017 00:00:00.

-------------------------- EXAMPLE 5 --------------------------

PS C:\>.\Get-AppLockerEvents.ps1 -Allowed -Objects | Group-Object PublisherName

Get allowed files from the EXE/DLL and MSI/Script logs on the local computer. Convert output into objects, group the objects according to the PublisherName field.

-------------------------- EXAMPLE 6 --------------------------

PS C:\>.\Get-AppLockerEvents.ps1 -NoOriginalPath -NoEventTime -NoEventTimeXL -NoUserSID -Objects | Where-Object { $\_.PublisherName.Length -le 1 } | ConvertTo-Csv -Delimiter "`t" -NoTypeInformation

Get warning and error events from the EXE/DLL and MSI/Script logs on the local computer, outputting only unsigned files. It converts output into objects, filters on PublisherName length (allowing up to a hyphen in length), then converts back to tab-delimited CSV.

-------------------------- EXAMPLE 7 --------------------------

PS C:\>$ev = .\Get-AppLockerEvents.ps1 -Objects

$ev | Select-Object UserName, MachineName -Unique | Sort-Object UserName, MachineName

$ev.FileExt | Sort-Object -Unique

Output a list of each combination of users and machines reporting events, and a list of all observed file extensions involved with events.

## Save-WEFEvents.ps1

Captures forwarded events to a CSV file with the timestamp embedded in the file name.

Intended to be executed on a Windows Event Collector server.

***Syntax***

Save-WEFEvents.ps1 [[-rootdir] <String>] [[-daysBack] <Int32>] [[-label] <String>] [<CommonParameters>]

***Parameters***

**-rootdir <String>**

Directory in which to create the CSV file.

**-daysBack <Int32>**

Number of days ago from which to start retrieving data. E.g., "-daysBack 5" pulls data from the last 5 days.

**-label <String>**

Name to insert into the filename (can be useful to distinguish sources).

## Generate-EventWorkbook.ps1

Produces a multi-tab Excel workbook containing summary and details of AppLocker events to support advanced analysis.

***Syntax***

Generate-EventWorkbook.ps1 [-AppLockerEventsCsvFile] <String> [-SaveWorkbook] [<CommonParameters>]

***Parameters***

**-AppLockerEventsCsvFile <String>**

Path to CSV file produced by Get-AppLockerEvents.ps1 or Save-WEFEvents.ps1, ideally without any attributes removed, but must contain at least these: MachineName, PublisherName, ProductName, GenericPath, GenericDir, FileName, FileType, Hash

**-SaveWorkbook [<SwitchParameter>]**

If set, saves workbook to same directory as input file with same file name and default Excel file extension.

***Description***

Converts the saved output from the Get-AppLockerEvents.ps1 or Save-WEFEvents.ps1 scripts to a multi-tab Excel workbook supporting numerous views of the data, including:

* Summary tab showing date/time ranges of the reported events and other summary information.
* List of machines reporting events, and the number of events per machine.
* List of publishers of signed files appearing in events, and the number of events per publisher.
* All combinations of publishers/products for signed files in events.
* All combinations of publishers/products and generic file paths ("generic" meaning that user-specific paths are replaced with %LOCALAPPDATA%, %USERPROFILE%, etc., as appropriate).
* Paths of unsigned files, with filename alone, file type, and file hash.
* Files grouped by user.
* Full details from Get-AppLockerEvents.ps1.

These separate tabs enable quick determination of the files running afoul of AppLocker rules and help quickly determine whether/how to adjust the rules.

# Appendix B: CustomizationInputs and MergeRules-\* reference

To incorporate customizations into the policies it builds, the Create-Policies.ps1 script runs scripts in the CustomizationInputs directory and reads XML files in the MergeRules-\* directories. You can edit the CustomizationInputs scripts and copy XML files into the MergeRules-Static directory to control these customizations. Create-Policies.ps1 expects some of the scripts to return an array of strings; from others it expects an array of hash tables. This appendix describes each of those scripts and how to edit them. It also describes how Create-Policies.ps1 uses the MergeRules-Dynamic and MergeRules-Static directories so that you can specify additional customizations.

You can get the same information for the CustomizationInputs scripts from within PowerShell using Get-Help; for example:

Get-Help .\CustomizationInputs\GetExeFilesToBlacklist.ps1 -Detailed

## GetExeFilesToBlacklist.ps1

Script used by Create-Policies.ps1 to identify EXE files that should be disallowed by AppLocker for non-admin use. Can be edited if necessary.

***Description***

This script outputs a list of file paths under %windir% that need to be specifically disallowed by whitelisting rules. The list of files is consumed by Create-Policies.ps1, which builds the necessary AppLocker rules to block them. You can edit this file as needed for your environment, although it is recommended that none of the programs identified in this script be removed.

Note: the solution also blocks the loading of PowerShell v2 modules - these blocks are hardcoded into the base XML file. This module as currently designed can block only EXE files, not DLLs.

http://www.leeholmes.com/blog/2017/03/17/detecting-and-preventing-powershell-downgrade-attacks/

## GetSafePathsToAllow.ps1

Customizable script used by Create-Policies.ps1 that produces a list of additional "safe" paths to allow for non-admin execution.

***Description***

This script outputs a simple list of directories that can be considered "safe" for non-admins to execute programs from. The list is consumed by Create-Policies.ps1, which incorporates the paths into AppLocker rules allowing execution of EXE, DLL, and Script files.

NOTE: DIRECTORY/FILE PATHS IDENTIFIED IN THIS SCRIPT MUST NOT BE WRITABLE BY NON-ADMIN USERS!!!

You can edit this file as needed for your environment.

Note that each directory name must be followed by \\*, as in these examples:

"C:\ProgramData\App-V\\*"

"\\MYSERVER\Apps\\*"

Individual files can be allowed by path, also. Do not end those with "\\*"

Specify paths using only fixed local drive letters or UNC paths. Do not use mapped drive letters or SUBST drive letters, as the user can change their definitions. If X: is mapped to the read-only \\MYSERVER\Apps file share, and you allow execution in \\MYSERVER\Apps\\*, the user can run MyProgram.exe in that share whether it is referenced as \\MYSERVER\Apps\MyProgram.exe or as X:\MyProgram.exe. Similarly, AppLocker does the right thing with SUBSTed drive letters.

## HashRuleData.ps1

Script used to define hash rules without direct access to the files.

***Description***

This script outputs zero or more hash tables containing information to define hash rules. It supports creating hash rules based on AppLocker event data rather than on direct access to the files.

Each hash table must have each of the following properties:

* RuleCollection
* RuleName
* RuleDesc
* HashVal
* FileName

NOTES:

* RuleCollection must be one of "Exe", "Dll", "Script", or "Msi", and is CASE-SENSITIVE.
* HashVal must be "0x" followed by 64 hex digits (SHA256 hash).

Example:

@{

RuleCollection = "Script";

RuleName = "Contoso Products: DoGoodStuff.cmd - HASH RULE";

RuleDesc = "Identified in: %LOCALAPPDATA%\TEMP\DoGoodStuff.cmd";

HashVal = "0x4CA1CD60FBFBA42C00EA6EA1B56BEFE6AD90FE0EFF58285A75D77B515D864DAE";

FileName = "DoGoodStuff.cmd"

}

## KnownAdmins.ps1

Outputs a list of known administrative users or groups that should be ignored when scanning for "user-writable" directories.

***Description***

Outputs a list of zero or more administrative users or groups that Enum-WritableDirs.ps1 does not know about (e.g., custom domain or local groups or users), one to a line.

The script framework scans for "user-writable" directories, looking for "write" permissions and ignoring permissions granted to "known administrative" users and groups. The framework might fail to recognize custom domain groups and (in some cases) local user accounts as administrative. This script enables adding those entities to the list of known administrative users/groups.

Output one entity name or SID per line.

Examples where this might be needed:

* Custom domain groups that have administrative rights.
* On Azure Active Directory joined systems, enumeration of BUILTIN\Administrators might not work correctly - might need to enumerate administrative accounts explicitly.

Examples:

"DESKTOP-7TPCJ7J\renamedAdmin"

"CONTOSO\SCCM-Admins"

## TrustedSigners.ps1

Customizable script used by Create-Policies.ps1 that identifies publishers or publisher+product/file combinations to trust.

***Description***

TrustedSigners.ps1 outputs a sequence of hash tables that specify a label, and either a literal publisher name, or a path to a signed file to use as an example.

Each hash table has a "label" property that is incorporated into the rule name and description.

Each hash table also has either a "PublisherName" or an "exemplar" property:

* "PublisherName" is a literal canonical name identifying a publisher to trust.  
  When using PublisherName, you can also add optional properties:
  + "ProductName", to restrict trust just to that product by that publisher; with "ProductName" you can also add "BinaryName" to restrict to a specific internal file name, and optionally then "FileVersion" as well to specify a minimum allowed file version. When using BinaryName, you should also specify an explicit RuleCollection, to reduce the number of rules. (E.g., no sense in having a Script rule allowing "MSVCP80.DLL".)
  + "RuleCollection", to apply the trust only within a single RuleCollection. RuleCollection must be one of "Exe", "Dll", "Script", or "Msi", and it is CASE-SENSITIVE.
* "exemplar" is the path to a signed file; the publisher to trust is extracted from that signature. When using exemplar, you can also add an optional "useProduct" boolean value indicating whether to restrict publisher trust only to that file's product name. If "useProduct" is not specified, all files signed by the publisher are trusted.

Examples showing possible combinations:

# Trust everything by a specific publisher

@{

label = "Trust all Contoso";

PublisherName = "O=CONTOSO, L=SEATTLE, S=WASHINGTON, C=US";

}

# Trust all DLLs by a specific publisher

@{

label = "Trust all Contoso DLLs";

PublisherName = "O=CONTOSO, L=SEATTLE, S=WASHINGTON, C=US";

RuleCollection = "Dll";

}

# Trust a specific product published by a specific publisher

@{

label = "Trust all CUSTOMAPP files published by Contoso";

PublisherName = "O=CONTOSO, L=SEATTLE, S=WASHINGTON, C=US";

ProductName = "CUSTOMAPP";

}

# Trust any version of a specific signed file by a specific publisher/product

# RuleCollection must be one of Exe, Dll, Script, or Msi, and is CASE-SENSITIVE

@{

label = "Trust Contoso's SAMPLE.DLL in CUSTOMAPP";

PublisherName = "O=CONTOSO, L=SEATTLE, S=WASHINGTON, C=US";

ProductName = "CUSTOMAPP";

BinaryName = "SAMPLE.DLL";

FileVersion = "10.0.15063.0";

RuleCollection = "Dll";

}

# Trust everything signed by the same publisher as the exemplar file (Autoruns.exe)

@{

label = "Trust the publisher of Autoruns.exe";

exemplar = "C:\Program Files\Sysinternals\Autoruns.exe";

}

# Trust everything with the same publisher and product as the exemplar file (LuaBuglight.exe)

@{

label = "Trust everything with the same publisher and product as LuaBuglight.exe";

exemplar = "C:\Program Files\Utils\LuaBuglight.exe";

useProduct = $true

}

## TrustedSigners-MsvcMfc.ps1

Script designed to be dot-sourced into TrustedSigners.ps1 that supports the creation of publisher rules for observed MSVC\*.DLL and MFC\*.DLL files.

***Description***

There are already MSVC\* and MFC\* DLLs in Windows - this script also allows redistributable DLLs that often ship with other products and are installed into user-writable directories. This output allows any version of signed MSVC\* or MFC\* DLLs that shipped with a known version of Visual Studio. This is not the same as allowing anything signed by Microsoft or is part of Visual Studio - just the runtime library support DLLs.

This file can be updated as additional MSVC\* and MFC\* DLLs appear in event logs when observed executing from user-writable directories. Add more files as they are identified.

See TrustedSigners.ps1 for details about how this input is used.

## UnsafePathsToBuildRulesFor.ps1

Customizable script used by Create-Policies.ps1 that identifies user-writable paths containing files that need to be allowed to execute.

***Description***

This script outputs a sequence of hash tables that identify user-writable files or directory paths containing content that users must be allowed to execute. (The scripts favor publisher rules over hash rules.)

Each hash table must include "label" and "paths" properties, with additional optional properties.

Hash table properties:

|  |  |
| --- | --- |
| label | REQUIRED; incorporated into rules' names and descriptions. |
| paths | REQUIRED; identifies one or more paths (comma separated if more than one). If a path is a directory, rules are generated for the existing files in that directory. If a path is to a file, a rule is generated for that file. |
| pubRuleGranularity | OPTIONAL; specifies granularity of publisher rules.  If specified, must be one of the following:   * pubOnly - lowest granularity: Publisher rules specify publisher only * pubProduct - Publisher rules specify publisher and product * pubProductBinary - (default) Publisher rules specify publisher, product, and binary name * pubProdBinVer - highest granularity: Publisher rules specify publisher, product, binary name, and minimum version.   Microsoft-signed Windows and Visual Studio files are always handled at a minimum granularity of "pubProductBinary"; other Microsoft-signed files are handled at a minimum granularity of "pubProduct". |
| JSHashRules | OPTIONAL; if specified and set to $true, generates hash rules for unsigned .js files; otherwise, doesn't generate them. |
| noRecurse | OPTIONAL; if specified, rules are generated only for the files in the specified directory or directories. Otherwise, rules are also generated for files in subdirectories of the specified directory or directories. |
| enforceMinVersion | DEPRECATED and OPTIONAL. pubruleGranularity takes precedence if specified.  Otherwise, setting to $false equivalent to pubruleGranularity = pubProductBinary;  setting to $true equivalent to pubruleGranularity = pubProdBinVer. |

Examples of valid hash tables:

# Search one directory and its subdirectories for files to generate rules for.

# Default granularity for publisher rules: create a separate rule for each file but allow any file version.

@{

label = "OneDrive";

paths = "$env:LOCALAPPDATA\Microsoft\OneDrive";

}

# Search one directory and its subdirectories for files to generate rules for.

# Generated publisher rules contain only publisher and product names.

# (Note that some Microsoft-signed files will also include binary name.)

@{

label = "OneDrive";

paths = "$env:LOCALAPPDATA\Microsoft\OneDrive";

pubruleGranularity = "pubProduct";

}

# Search two separate directory structures for files to generate rules for, plus one explicitly-identified file.

@{

label = "ContosoIT";

paths = "$env:LOCALAPPDATA\Programs\MyContosoIT\Helper",

"C:\ProgramData\COntosoIT\ContosoIT System Health Client",

"$env:LOCALAPPDATA\TEMP\CORPSEC\ITGSECLOGONGPEXEC.EXE"

}

# Generate rules for three distinct files; do not recurse subdirectories looking for additional matches.

@{

label = "Custom backup scripts";

paths = "C:\Backups\MyBackup.vbs",

"C:\Backups\MyPersonalBackup.vbs",

"C:\Backups\Exports\RegExport.1.cmd";

noRecurse = $true

}

## The MergeRules-Dynamic and MergeRules-Static directories

The MergeRules-\* directories contain XML files representing AppLocker policy fragments. Create-Policies.ps1 merges each of the \*.xml files it finds in these two directories into the policies it builds. Create-Policies.ps1 clears and rebuilds the content of the MergeRules-Dynamic directory each time it runs, but treats MergeRules-Static as read-only.

### MergeRules-Dynamic

Each time Create-Policies.ps1 runs, it first deletes any XML files in the MergeRules-Dynamic directory. It then processes the hash tables returned by UnsafePathsToBuildRulesFor.ps1 and creates up to two XML files for each one, using the hash table’s *label* attribute in the file name. For example, if a label value is “OneDrive,” CreatePolicies.ps1 creates MergeRules-Dynamic\OneDrive Publisher Rules.xml and MergeRules-Dynamic\OneDrive Hash Rules.xml. The XML files each contain an AppLocker policy fragment containing publisher or hash rules for the files that Create-Policies.ps1 identified in the paths specified by the hash table.

If TrustedSigners.ps1 returns any entries, Create-Policies.ps1 generates a policy fragment containing corresponding publisher rules into TrustedSigners.xml in the MergeRules-Dynamic directory. Similarly, if HashRulesData.ps1 returns any entries, Create-Policies.ps1 creates a policy fragment containing hash rules into ExtraHashRules.xml.

The fragments derived from UnsafePathsToBuildRulesFor.ps1 depend on files being available in the unsafe paths to pull information from. If you can’t count on those files always being available (for example, if you want to be able to build the same rule set on another system), save the resulting XML files. You can move them into the MergeRules-Static directory and then you can recreate the same rules without having to scan files.

By contrast, the policy fragments in ExtraHashRules.xml are derived entirely from data within HashRulesData.ps1. Similarly, the fragments in TrustedSigners.xml derive entirely from data in TrustedSigners.ps1 (unless you’re using its “exemplar” attribute, which points to an external file). If you want to move TrustedSigners.xml and ExtraHashRules.xml into the MergeRules-Static directory, you can, but you don’t need to.

### MergeRules-Static

Create-Policies.ps1 treats MergeRules-Static as read-only. It reads all XML files in the directory and merges their content into the policies that it builds, but it does not modify the directory’s contents in any way. You can copy any AppLocker policy fragment files you want – including Allow or Deny rules – into this directory and Create-Policies.ps1 will merge them into the final policy.

In addition to copying files from the MergeRules-Dynamic directory, you can use the BuildRulesForFilesInWritableDirectories.ps1 script described in the “Support scripts reference” appendix. On the more elaborate end, you can create rules using the AppLocker GUI wizard, export the resulting rules in XML form and hand-edit them.

For example, here’s how you can block specific “Packaged apps” (modern apps, Store apps, UWP apps – I’m pretty sure our marketing folks get bonuses based on how often they rename things):

* In the AppLocker GUI, navigate to the Packaged app Rules node;
* Create “Deny” rules for the packaged apps that you want to disallow;
* Right-click on the AppLocker icon in the console tree, choose “Export Policy…,” and save the AppLocker policy to an XML file.
* Edit the XML file and remove all the RuleCollection elements other than the “Appx” one.
* Copy the edited XML file into your MergeRules-Static directory.

# Appendix C: Support scripts reference

As the directory name suggests, files in the AaronLocker Support directory are there to support the main scripts. You don’t need to use them directly, but you can if you have a particular need:

* **Set-OutputEncodingToUnicode.ps1** changes the session’s character encoding to Unicode.
* **DownloadAccesschk.ps1** downloads AccessChk.exe to the main AaronLocker directory.
* **Enum-WritableDirs.ps1** lists the user-writable subdirectories in a path that you point to.
* **BuildRulesForFilesInWritableDirectories.ps1** creates a single AppLocker policy fragment XML file containing publisher and hash rules for the files in a path you point to.
* **ExportPolicy-ToCsv.ps1** takes a given AppLocker policy and converts it to a more readable CSV. (It is primarily used by ExportPolicy-ToExcel.ps1.)

## Set-OutputEncodingToUnicode.ps1

Sets the output encoding for the current session to Unicode, so that piped output retains Unicode encoding.

That’s it. No parameters or switches.

## DownloadAccesschk.ps1

Downloads Sysinternals AccessChk.exe to the main AaronLocker directory.

## Get-AaronLockerTimestamp.ps1

Gets custom timestamp field from AaronLocker-generated AppLocker rule set.

***Syntax***

Get-AaronLockerTimestamp.ps1 [-Local]

Get-AaronLockerTimestamp.ps1 [-AppLockerXML <String>]

***Parameters***

**-Local [<SwitchParameter>]**

If this switch is specified, the script processes the computer's local AppLocker policy.

If no parameters are specified or this switch is set to -Local:$false, the script processes the computer's effective AppLocker policy.

**-AppLockerXML <String>**

If this parameter is specified, AppLocker policy is read from the specified exported XML policy file.

***Description***

Retrieves an AppLocker policy, and reports the AaronLocker-generated timestamp, if found.

AaronLocker inserts a "timestamp" rule that shows when the rule set was generated and helps associate it with a rule file with the same timestamp.

This script can inspect local policy, effective policy, or an AppLocker policy XML file.

The AaronLocker-generated timestamp is stored in the name and description of an Exe "deny" hash rule with a bogus hash value and applied to the "CREATOR OWNER" user. Because "CREATOR OWNER" never appears in a user's access token, the rule will never be applied. The "Deny" and "CREATOR OWNER" attributes make it stand out and easily visible in an AaronLocker rule set.

***Example***

PS C:\>.\Support\Get-AaronLockerTimestamp.ps1

Gets the custom timestamp field from the computer's effective AppLocker policy.

## Enum-WritableDirs.ps1

Enumerates "user-writable" subdirectories.

***Syntax***

Enum-WritableDirs.ps1 [-RootDirectory] <String> [-ShowGrantees] [-DontFilterNTService] [-OutputXML] [[-KnownAdmins] <String[]>] [<CommonParameters>]

***Parameters***

**-RootDirectory <String>**

The starting directory for the permission enumeration.

**-ShowGrantees [<SwitchParameter>]**

If set, output includes the names of the non-admin entities that have write permissions

**-DontFilterNTService [<SwitchParameter>]**

By default, this script ignores access granted to NT SERVICE\ accounts (SID beginning with S-1-5-80-). If this switch is set, this script does not ignore that access, except for access granted to NT SERVICE\TrustedInstaller.

**-OutputXML [<SwitchParameter>]**

If set, output is formatted as XML.

**-KnownAdmins <String[]>**

Optional: additional list of known administrative users and groups.

***Description***

Enumerates subdirectories that are writable by accounts other than a set of known admin or admin-equivalent entities (including members of the local Administrators group). The goal is to list user-writable directories in which end user program execution should be disallowed via AppLocker.

You should run this script with administrative rights to avoid access-denied errors.

NOTE: Requires Sysinternals AccessChk.exe:

<https://technet.microsoft.com/sysinternals/accesschk>

<https://download.sysinternals.com/files/AccessChk.zip>

<https://live.sysinternals.com/accesschk.exe>

or run Support\DownloadAccesschk.ps1, which downloads AccessChk.exe to the main AaronLocker directory.

NOTE: Requires Windows PowerShell 5.1 or newer (relies on Get-LocalGroup and Get-LocalGroupMember cmdlets).

Note: this script does not discover user-writable files. A user-writable file in a non-writable directory presents a similar risk, as a non-admin can overwrite it with arbitrary content and execute it.

***Examples***

-------------------------- EXAMPLE 1 --------------------------

PS C:\>.\Enum-WritableDirs.ps1 C:\Windows\System32

Output:

C:\Windows\System32\FxsTmp

C:\Windows\System32\Tasks

C:\Windows\System32\Com\dmp

C:\Windows\System32\Microsoft\Crypto\RSA\MachineKeys

C:\Windows\System32\spool\PRINTERS

C:\Windows\System32\spool\SERVERS

C:\Windows\System32\spool\drivers\color

C:\Windows\System32\Tasks\Microsoft IT Diagnostics Utility

C:\Windows\System32\Tasks\Microsoft IT VPN

C:\Windows\System32\Tasks\WPD

C:\Windows\System32\Tasks\Microsoft\Windows\RemoteApp and Desktop Connections Update

C:\Windows\System32\Tasks\Microsoft\Windows\SyncCenter

C:\Windows\System32\Tasks\Microsoft\Windows\WCM

C:\Windows\System32\Tasks\Microsoft\Windows\PLA\System

-------------------------- EXAMPLE 2 --------------------------

PS C:\>.\Enum-WritableDirs.ps1 C:\Windows\System32 -ShowGrantees

Output:

C:\Windows\system32\FxsTmp

BUILTIN\Users

C:\Windows\system32\Tasks

NT AUTHORITY\Authenticated Users

C:\Windows\system32\Com\dmp

BUILTIN\Users

C:\Windows\system32\Microsoft\Crypto\RSA\MachineKeys

Everyone

C:\Windows\system32\spool\PRINTERS

BUILTIN\Users

C:\Windows\system32\spool\SERVERS

BUILTIN\Users

C:\Windows\system32\spool\drivers\color

BUILTIN\Users

C:\Windows\system32\Tasks\Microsoft IT Diagnostics Utility

NT AUTHORITY\Authenticated Users

C:\Windows\system32\Tasks\Microsoft IT VPN

NT AUTHORITY\Authenticated Users

C:\Windows\system32\Tasks\WPD

NT AUTHORITY\Authenticated Users

aaronmar5\aaronmaradmin

C:\Windows\system32\Tasks\Microsoft\Windows\RemoteApp and Desktop Connections Update

NT AUTHORITY\Authenticated Users

C:\Windows\system32\Tasks\Microsoft\Windows\SyncCenter

BUILTIN\Users

C:\Windows\system32\Tasks\Microsoft\Windows\WCM

BUILTIN\Users

C:\Windows\system32\Tasks\Microsoft\Windows\PLA\System

Everyone

-------------------------- EXAMPLE 3 --------------------------

PS C:\>$x = [xml](.\Enum-WritableDirs.ps1 C:\Windows\System32 -ShowGrantees -OutputXML)

PS C:\>$x.root.dir | Sort-Object name

Output:

name Grantee

---- -------

C:\Windows\System32\Com\dmp BUILTIN\Users

C:\Windows\System32\FxsTmp BUILTIN\Users

C:\Windows\System32\Microsoft\Crypto\RSA\MachineKeys Everyone

C:\Windows\System32\spool\drivers\color BUILTIN\Users

C:\Windows\System32\spool\PRINTERS BUILTIN\Users

C:\Windows\System32\spool\SERVERS BUILTIN\Users

C:\Windows\System32\Tasks NT AUTHORITY\Authenticated Users

C:\Windows\System32\Tasks\Microsoft IT Diagnostics Utility NT AUTHORITY\Authenticated Users

C:\Windows\System32\Tasks\Microsoft IT VPN NT AUTHORITY\Authenticated Users

C:\Windows\System32\Tasks\Microsoft\Windows\PLA\System Everyone

C:\Windows\System32\Tasks\Microsoft\Windows\RemoteApp and Desktop Connection... NT AUTHORITY\Authenticated Users

C:\Windows\System32\Tasks\Microsoft\Windows\SyncCenter BUILTIN\Users

C:\Windows\System32\Tasks\Microsoft\Windows\WCM BUILTIN\Users

C:\Windows\System32\Tasks\WPD {NT AUTHORITY\Authenticated Users, vm-t2408\admin}

## BuildRulesForFilesInWritableDirectories.ps1

Builds tightly-scoped but forward-compatible AppLocker rules for files in user-writable directories. The rules are intended to be merged into a larger set using Create-Policies.ps1 in the root directory.

***Syntax***

BuildRulesForFilesInWritableDirectories.ps1 -FileSystemPaths <String[]> [-RecurseDirectories] [-PubRuleGranularity <String>] [-JSHashRules] -OutputPubFileName <String> -OutputHashFileName <String> [-RuleNamePrefix <String>] [<CommonParameters>]

BuildRulesForFilesInWritableDirectories.ps1 -FileOfFileSystemPaths <String> [-RecurseDirectories] [-PubRuleGranularity <String>] [-JSHashRules] -OutputPubFileName <String> -OutputHashFileName <String> [-RuleNamePrefix <String>] [<CommonParameters>]

***Parameters***

**-FileSystemPaths <String[]>**

An array of file paths and/or directory paths to scan. The array can be a comma-separated list of file system paths. Either FileSystemPaths or FileOfFileSystemPaths must be specified.

**-FileOfFileSystemPaths <String>**

The name of a file containing a list of file paths and/or directory paths to scan; one path to a line. Either FileSystemPaths or FileOfFileSystemPaths must be specified.

**-RecurseDirectories [<SwitchParameter>]**

If this switch is specified, scanning of directories includes subdirectories; otherwise, only files in the named directory are scanned.

**-PubRuleGranularity <String>**

Optional parameter to specify the granularity of generated publisher rules. If specified, must be one of the following:  
\* pubOnly - lowest granularity: Publisher rules specify publisher only  
\* pubProduct - Publisher rules specify publisher and product  
\* pubProductBinary - (default) Publisher rules specify publisher, product, and binary name  
\* pubProdBinVer - highest granularity: Publisher rules specify publisher, product, binary name, and minimum version.

Note that Microsoft-signed Windows and Visual Studio files are always handled at a minimum granularity of "pubProductBinary"; other Microsoft-signed files are handled at a minimum granularity of "pubProduct".

**-JSHashRules <String>**

If this switch is specified, generates hash rules for unsigned .js files; otherwise, doesn't.

**-OutputPubFileName <String>**

Required: the name/path of the XML output file containing the generated publisher rules.

**-OutputHashFileName <String>**

Required: the name/path of the XML output file containing the generated hash rules.

**-RuleNamePrefix <String>**

Optional: If specified, all rule names begin with the specified RuleNamePrefix.

***Description***

This script takes a list of one or more file system objects (files and/or directories) and generates rules to allow execution of the corresponding files. Rule files generated with this script can be incorporated into comprehensive rule sets using Create-Policies.ps1 in the root directory.

Publisher rules are generated where possible:

* Publisher rules restrict to a specific publisher, product name, binary name, and minimum file version. Optionally, less-granular rules can be generated (e.g., publisher only, publisher/product only, etc.)
* Redundant rules are removed; if multiple versions of a specific file are found, the rule allows execution of the lowest-identified version or above.

Hash rules are generated when publisher rules cannot be created.

The script creates rule names and descriptions designed for readability in the Security Policy editor. The RuleNamePrefix option enables you to give each rule in the set a common prefix (e.g., "OneDrive") to make the source of the rule more apparent and so that related rules can be grouped alphabetically by name. The rules' EnforcementMode is left NotConfigured. (Create-Policies.ps1 takes care of setting EnforcementMode in the larger set.) (Note that the New-AppLockerPolicy's -Optimize switch "overoptimizes," allowing any file name within a given publisher and product name. Not using that.)

File system objects can be identified on the command line with -FileSystemPaths, or listed in a file (one object per line) referenced by -FileOfFileSystemObjects.

This script determines whether each object is a file or a directory. For directories, this script enumerates and identifies EXE, DLL, and Script files based on file extension. Subdirectories are scanned if the -RecurseDirectories switch is specified on the command line.

The intent of this script is to create fragments of policies that can be incorporated into a "master" policy in a modular way. For example, create a file representing the rules needed to allow OneDrive to run, and separate files for LOB apps. If/when the OneDrive rules need to be updated, they can be updated in isolation and those results incorporated into a new master set.

***Examples***

PS C:\>.\BuildRulesForFilesInWritableDirectories.ps1 -FileSystemPaths $env:LOCALAPPDATA\Microsoft\OneDrive -RecurseDirectories -RuleNamePrefix OneDrive -OutputPubFileName ..\WorkingFiles\OneDrivePubRules.xml -OutputHashFileName ..\WorkingFiles\OneDriveHashRules.xml

Scans the OneDrive directory and subdirectories in the current user's profile. All generated rule names will begin with "OneDrive". The generated publisher rules are written to ..\WorkingFiles\OneDrivePubRules.xml and the generated hash rules to ..\WorkingFiles\OneDriveHashRules.xml. Publisher rules created with default granularity (one rule per file).

## ExportPolicy-ToCsv.ps1

Turn AppLocker policy into more human-readable CSV.

***Syntax***

ExportPolicy-ToCsv.ps1 [[-AppLockerPolicyFile] <String>] [-Local] [[-linebreakSeq] <String>] [<CommonParameters>]

***Parameters***

**-AppLockerPolicyFile <String>**

If this optional string parameter is specified, AppLocker policy is read from the specified XML file.

**-Local [<SwitchParameter>]**

If this switch is specified, the script processes the current computer's local policy.

**-linebreakSeq <String>**

If this optional string parameter is specified, CRLF and LF sequences in attribute values are replaced with the specified sequence. "^|^" is the default.

***Description***

Script reads AppLocker policy from local policy, effective policy, or an XML file, and renders it as a tab-delimited CSV that can be pasted into Microsoft Excel, with easy sorting and filtering.

If neither -AppLockerPolicyFile <path> or -Local is specified, the script processes the current computer's effective policy.

If -linebreakSeq is not specified, CRLF and LF sequences in attribute values are replaced with "^|^". The linebreak sequence can be replaced after importing results into Excel (in the Find/Replace dialog, replace the sequence with Ctrl+Shift+J).

***Examples***

PS C:\>ExportPolicy-ToCsv.ps1 | clip.exe

Renders effective AppLocker policy to tab-delimited CSV and writes that output to the clipboard using the built-in Windows clip.exe utility.

Paste the output directly into an Excel spreadsheet, replace "^|^" with Ctrl+Shift+J, add filtering, freeze the top row, and autosize.

## Other files

**SupportFunctions.ps1** contains global support functions. It’s intended to be dot-sourced into other scripts and not run directly. Its contents include:

* Functions to save XML consistently as Unicode:
  + SaveXmlDocAsUnicode([System.Xml.XmlDocument] $xmlDoc, [string] $xmlFilename)
  + SaveAppLockerPolicyAsUnicodeXml([Microsoft.Security.ApplicationId.PolicyManagement.PolicyModel.AppLockerPolicy]$ALPolicy, [string]$xmlFilename)
* Functions to create Excel spreadsheets/workbooks:
  + CreateExcelApplication()
  + ReleaseExcelApplication()
  + SelectFirstWorksheet()
  + SaveWorkbook([string]$filename)
  + AddNewWorksheet([string]$tabname)
  + AddWorksheetFromText([string[]]$text, [string]$tabname)
  + AddWorksheetFromCsvFile([string]$filename, [string]$tabname, [string]$CrLfEncoded)
  + AddWorksheetFromCsvData([string[]]$csv, [string]$tabname, [string]$CrLfEncoded)
  + CreateExcelFromCsvFile([string]$filename, [string]$tabname, [string]$CrLfEncoded, [string]$saveAsName)
* Function to determine whether a file is a Win32 EXE, a Win32 DLL, or neither:
  + IsWin32Executable([string]$filename)
* Global variables defining known file extensions.

**Config.ps1** defines variables for path names and other configuration settings, and loads global support functions. It is intended to be dot-sourced into other scripts, and not run directly. The variable $rootDir must already have been set prior to calling this script.

**DefaultRulesWithPlaceholders.xml** is what Create-Policies.ps1 starts from. It includes the rules for Administrators, the initial rules for “Everyone” (i.e., non-administrators) with placeholders to be filled in for exceptions. It also includes the rules that block PowerShell version 2. You should never need to edit this file.

# Appendix D: PowerShell and whitelisting

Although many people think of it as just a scripting platform, Microsoft Windows PowerShell is a full-featured programming platform. You can create binary data structures, call Win32 APIs, invoke COM objects and .NET interfaces, and even generate native executable code in memory and then execute it. If AppLocker policy allows PowerShell.exe and its DLLs, AppLocker cannot limit what kind of code that process then executes. Obviously, this is problematic for any whitelisting solution, and prior to PowerShell version 5 the only option was to block PowerShell entirely.

PowerShell version 5 introduced whitelisting awareness that enforces PowerShell’s ConstrainedLanguage mode. When PowerShell starts, it determines whether a whitelisting solution is operative and limiting script execution for the current user. If not, PowerShell runs normally, in “FullLanguage” mode. If whitelisting is in effect, then that instance of PowerShell runs in ConstrainedLanguage mode.

In ConstrainedLanguage mode, you cannot create data structures or invoke .NET, COM, or Win32 interfaces. You can still run “safe” commands and cmdlets and also use pipelines. If you run scripts allowed by AppLocker script rules – whether through Path, Publisher, or Hash rules – those scripts run in FullLanguage mode. If you run a script that is not covered by an AppLocker “allow” rule, the script will still run, but in ConstrainedLanguage mode. (In my opinion, this is a great balance. Users can still benefit from the basic features of PowerShell, but can’t run arbitrary code.)

Spurious AppLocker event log entries are one of the side effects of PowerShell’s detecting whether AppLocker script rules are in effect. PowerShell generates randomly-named script files in the user’s temp directory and then invokes APIs to determine whether AppLocker allows them. Those APIs also log results to the AppLocker “MSI and Script” event log; in “enforce” mode you will see error events every time the user starts PowerShell. You can safely ignore these events, and the AaronLocker event-log scripts filter out these events by default.

For backward compatibility, PowerShell provides the ability to run earlier PS engine versions by starting PowerShell.exe with the -Version command-line option. Because PowerShell’s whitelisting awareness was not backported to earlier PowerShell versions, this would create a whitelisting-bypass vulnerability. To mitigate this exposure, AaronLocker explicitly blocks the PowerShell DLLs that support all earlier versions. On a default installation of Windows 7 without PowerShell upgraded to version 5.x, this also has the effect of blocking PowerShell entirely.

Review these references for more information about PowerShell’s whitelisting awareness and Constrained Language Mode:

**PowerShell <3 the Blue Team**<https://blogs.msdn.microsoft.com/powershell/2015/06/09/powershell-the-blue-team/>

**About Language Modes**<https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_language_modes?view=powershell-5.1>

**PowerShell Constrained Language Mode**  
<https://blogs.msdn.microsoft.com/powershell/2017/11/02/powershell-constrained-language-mode/>

*[[[ TODO: Discuss PS v5.1 vs. 6 ]]]*

# Appendix E: Troubleshooting tips

This section contains a growing list of troubleshooting tips to help resolve problems you might run into with AaronLocker.

## PowerShell script errors

Errors that you might see running AaronLocker scripts and the causes of those errors:

### *scriptname*.ps1 cannot be loaded because running scripts is disabled on this system.

You need to set the execution policy to allow scripts. Use a command such as Set-ExecutionPolicy RemoteSigned -Force in an administrative PowerShell session. Other sessions need to be restarted to pick up the new execution policy.

### *scriptname*.ps1 cannot be loaded. The file *scriptname*.ps1 is not digitally signed. You cannot run this script on the current system.

You have set the script execution policy to RemoteSigned and the scripts are either on a remote system or have the “mark of the web” on them. If they are on the local file system, make sure to “unblock” the zip file before extracting the files from them. You can also use the PowerShell Unblock-File cmdlet or the Sysinternals streams.exe tool to remove the alternate data streams on the files.

### *scriptname*.ps1 cannot be loaded because its operation is blocked by software restriction policies, such as those created by using Group Policy.

You have applied AppLocker enforce-mode rules and you are running AaronLocker scripts as a non-administrative user in a location that has not been whitelisted. Some of your options are to move the scripts to an approved location (e.g., under ProgramFiles), to run the scripts from an administrative PowerShell session, or to remove AppLocker enforcement.

1. Windows Defender Application Control was formerly known as “Device Guard.” [↑](#footnote-ref-2)
2. Only when operations are performed beyond of the control of the operating system can meaningful restraints begin to be placed on administrators. Examples include operations performed by trusted platform modules (TPMs), smart cards, and Windows 10’s Virtualization Based Security upon which Credential Guard and HVCI are built. But even when some aspects of whitelisting are managed by such components (e.g., as WDAC offers) and administrative bypasses of whitelisting rules are more difficult to achieve, they aren’t impossible. [↑](#footnote-ref-3)
3. <https://apps.nsa.gov/iaarchive/library/ia-guidance/tech-briefs/application-whitelisting-using-microsoft-applocker.cfm>. Note that the server authentication certificate chains up to a US Dept. of Defense root certificate, which is not in the Windows Root Certificate Program and so is not trusted by default. To download the document, you might need to ignore browser warnings about the certificate or install the US DoD root certificate. [↑](#footnote-ref-4)
4. You can incorporate Packaged-app rules if you want. See “The MergeRules-Dynamic and MergeRules-Static directories” in Appendix B for details. [↑](#footnote-ref-5)
5. Version-resource information is a standard but optional feature of Portable Executable (PE) files including EXEs and DLLs. You can view version information with the Details tab in Explorer’s Properties dialogs, the Sysinternals SigCheck utility, or the old “filever” resource kit utility. For more information about version resources, see <https://docs.microsoft.com/en-us/windows/desktop/menurc/version-information>. [↑](#footnote-ref-6)
6. For executable and DLL rules, Windows uses the file’s Authenticode SHA256 hash; for other file types it uses the flat-file SHA256 hash. [↑](#footnote-ref-7)
7. The name “AaronLocker” was Chris (@appcompatguy) Jackson’s idea – not mine – and I resisted it for a long time. I finally gave in because I couldn’t come up with a better name. [↑](#footnote-ref-8)
8. See the appendix about PowerShell and whitelisting [↑](#footnote-ref-9)
9. See Get-AaronLockerTimestamp.ps1 in the “Support scripts reference” appendix for full details. [↑](#footnote-ref-10)
10. See the appendix about PowerShell for the details. [↑](#footnote-ref-11)
11. <https://technet.microsoft.com/sysinternals/accesschk>, or run Support\DownloadAccesschk.ps1, which downloads AccessChk.exe to the main AaronLocker directory. [↑](#footnote-ref-12)
12. You can use any text editor. Windows includes the Windows PowerShell ISE (powershell\_ise.exe) which is designed for PowerShell scripts. You can use the alias *ise* from the PowerShell console to open a file in the ISE, or right-click the file in Explorer and choose Edit from the context menu. [↑](#footnote-ref-13)
13. Administrative rights are recommended when scanning for user-writable directories, as administrative rights provide better visibility into those file-system permissions than non-admin accounts have. [↑](#footnote-ref-14)
14. For more information, see Jessica Payne’s blog post, *Monitoring what matters – Windows Event Forwarding for everyone (even if you already have a SIEM.)* at this URL: <https://blogs.technet.microsoft.com/jepayne/2015/11/23/monitoring-what-matters-windows-event-forwarding-for-everyone-even-if-you-already-have-a-siem/>

    Also see this page which covers Windows Event Collector scalability planning: <https://docs.microsoft.com/en-us/windows/security/threat-protection/use-windows-event-forwarding-to-assist-in-intrusion-detection> [↑](#footnote-ref-15)
15. AppLocker also has logs pertaining to packaged apps, but because AaronLocker does not restrict packaged apps, Get-AppLockerEvents.ps1 does not inspect those logs. [↑](#footnote-ref-16)
16. See the appendix about PowerShell and whitelisting. [↑](#footnote-ref-17)
17. <https://blogs.msdn.microsoft.com/dotnet/2013/08/06/got-a-need-for-speed-net-apps-start-faster/> [↑](#footnote-ref-18)
18. I’m using “ProgramFiles” here to refer to both the “Program Files” and “Program Files (x86)” directories. [↑](#footnote-ref-19)
19. An OOXML document can reference a template that contains macros. [↑](#footnote-ref-20)
20. Download SCT components here: <https://www.microsoft.com/en-us/download/details.aspx?id=55319>. The Office 2016 baseline is described here: <https://blogs.technet.microsoft.com/secguide/2018/02/13/security-baseline-for-office-2016-and-office-365-proplus-apps-final/> [↑](#footnote-ref-21)
21. The Microsoft Office Subject Interface Packages (SIPs) facilitate the signing of Office VBA projects: <https://www.microsoft.com/en-us/download/details.aspx?id=56617> [↑](#footnote-ref-22)
22. The Office 365 monthly channel now blocks Flash, Shockwave, and Silverlight in-process activation by default. [↑](#footnote-ref-23)