Inexorable PowerShell - A Red Teamer's Tale of Overcoming Simple AppLocker Policies

By sixdub

Published: 2014-12-02 · Archived: 2026-04-05 23:15:49 UTC

EDIT This repo has been renamed to PowerPick and added to the Veil-Framework's PowerTools. Find it HERE! See below for more edits. *EDIT*

Attackers have evolved to love PowerShell more than most defenders or system administrators. Tools like Powersploit', Veil Power*, and Nishang have become routine capabilities used by Red Teams, Pentesters, Evil attackers, and skiddies alike. With this evolution and overall consolidation of techniques into a single scripting language, surely defenders have found a proven method to prevent PowerShell execution? Surely Software Restriction Policies (SRP) or AppLocker can save the day? Don't be so sure…

Assessment after Assessment, I find that we can compromise a domain user, elevate local privileges, steal credentials, inject payloads, and escalate in the domain all using PowerShell. I have nightmares of the day someone effectively restricts PowerShell and some of the old school tactics must return. From my conversations with defenders or infosec junkies, awareness of these techniques is on the rise and people are finally starting to pay attention to the routine release of PowerShell tools to aid in offense. With that being said, until disabling PowerShell on unneeded systems becomes common practice in the trenches of commercial enterprise, attackers will still have an easy[ier] win. At this point, the restriction of PowerShell is unlikely to happen until the time/cost required to implement such defensives are minimized to a point where it can be realistically accomplished natively at scale.

For some previous research attackers' use of PowerShell:

- FireEye WhitePaper from Blackhat – Includes discussion of incident response with PowerShell. Awesome writeup! Props to these guys for taking a stab at defensive conversation in this arena. I hope to see some of this work recreated on an engagement some time.
- Crowdstrike Report on DeepPanda – Example of threat actor using PowerShell
- Weaponizing PowerShell – harmj0ys post on weaponizing PowerShell. Good write up on bypassing execution restrictions
- PowerShell Basics – Carlos Perez tutorials on PowerShell. Definitely worth the read
- Powersploit' Github – Essential for Offensive PowerShell users

In The Words of The Defenders – "Use Applocker"

Disclaimer: This is not intended to be a guide on AppLocker. Later you will see, this is a guide of what not to do! Also, I am not a Microsoft Certified Systems Engineer (MSCE) or any sort of Microsoft professional. There might be ways to properly configure AppLocker to prevent PowerShell… but they are not publicized enough if they exist! I googled as much as the average network defender would.

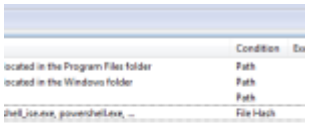
AppLocker is the Microsoft solution for “application control in the enterprise”. It is built to restrict unwanted software from being executed and provides a variety of methods to accomplish this. It allows you to specify policies that limit executables, DLLs, installers or scripts by path, hash, or publisher. The resulting policies are then pushed out by Group Policy and managed centrally.

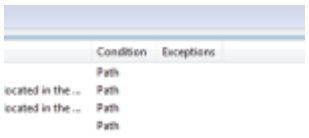
Obviously there is the regularly preached balance between usability and security but I find it important here to mention that the best setup possible with AppLocker would utilize a whitelist approach. Organizations could analyze their standard images and build policies to match this image. With that being said, the CEO wouldn’t be able to install their P2P software, the user wouldn’t be able to run World of Warcraft, and tech support would have an increase call volume by 3000% (Seriously hope to never work help desk… but I sure do appreciate the work they do!). For this reason, organizations still rely on blacklist based policies to prevent the use of net*.exe, cmd.exe and powershell.exe.

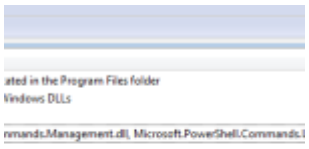
For the purpose of my demo, I intended to mimic an organization that used AppLocker in a black list fashion. My goal was to use AppLocker as much as possible to block PowerShell and test appropriate measures to get around the blacklist.

I tested with a Windows 7 system. I performed the following actions to attempt to setup and secure my test machine:

- Started the Application Identity service
- Added Executable Rules to deny by hash the following executables:
    - C:\windows\system32\WindowsPowerShellv1.0\powershell.exe
    - C:\windows\system32\WindowsPowerShellv1.0\powershell_ise.exe
    - C:\windows\Syswow64\WindowsPowerShellv1.0\powershell.exe
    - C:\windows\Syswow64\WindowsPowerShellv1.0\powershell_ise.exe
- Added Script Rules to deny by path: *.ps1*
- Enabled “Enforce” on DLL Rules (AppLocker->Properties)
- Added DLL Rules to deny by hash the following DLLs:
    - C:\Program Files\Reference Assemblies\Microsoft\WindowsPowerShellv1.0\Microsoft.PowerShell.Commands.Management.dll
    - C:\Program Files\Reference Assemblies\Microsoft\WindowsPowerShellv1.0\Microsoft.PowerShell.Commands.Utility.dll
    - C:\Program Files\Reference Assemblies\Microsoft\WindowsPowerShellv1.0\Microsoft.PowerShell.ConsoleHost.dll
    - C:\Program Files\Reference Assemblies\Microsoft\WindowsPowerShellv1.0\Microsoft.PowerShell.Security.dll
    - C:\Program Files\Reference Assemblies\Microsoft\WindowsPowerShellv1.0\System.Management.Automation.dll
- Tested to ensure the block worked! SUCCESS!







The Work Around – Lock Picking the AppLocker

I have recently gotten into the Offensive PowerShell world and have only tip toed at this point. I have heard a couple people discuss methods of bypassing the AppLocker rules with regards to PowerShell. **Personally, I give props to harmj0y for the inspiration and tips in the right direction. I also know many of the ideas and initial research discovering this technique came from darkoperator and a few others in the offensive PowerShell community**. Thanks to those who came before and paved the way. Would love to hear if you have better methods!

PowerShell provides backend access through the .NET framework and Windows Common Language Interface (CLI). Developers love the extensibility this provides and many examples exist online for how to use these features to create utilities to do fun sys-ad type things. Few people realized that the same code could be used to package PowerShell as a workaround for AppLocker restrictions.

Take a read of this for info on how to invoke PS scripts in C#.

First I must configure my project and add the Automation assembly as a reference:

- RC on References->Add Reference
- Browse and select: C:Program FilesReference AssembliesMicrosoftWindowsPowerShellv1.0System.Management.Automation.dll

In order to load from a resource, I must first create a resource:

- Open Resources.resx
- Add String Resource
- Name the string “Script”
- Set the value to “Get-Process” or whatever script you want to execute

After creating the project, I instantiate my Runspace and create the Pipeline that will be used to execute commands

```
//Init stuff
Runspace runspace = RunspaceFactory.CreateRunspace();
runspace.Open();
RunspaceInvoke scriptInvoker = new RunspaceInvoke(runspace);
Pipeline pipeline = runspace.CreatePipeline();
```

Next, I extract a PowerShell script from the resource section and add it to the pipeline. This could be changed to read a PowerShell script from the internet, from an encrypted string, etc. In a target environment, it would be important to obfuscate/encrypt the script to prevent AV or HIPS from triggering on the plaintext script. I have see on disk scripts get a Red Team caught every now and then….

```
//Add commands
string script = Properties.Resources.ResourceManager.GetString("Script");
pipeline.Commands.AddScript(script);
```

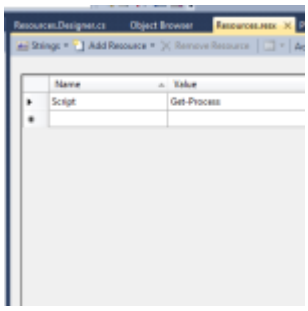
To retrieve the output from the script, I add the “Out-String” command onto the pipeline and grab the return objects. iterate over these objects with a string builder to return the final output string.

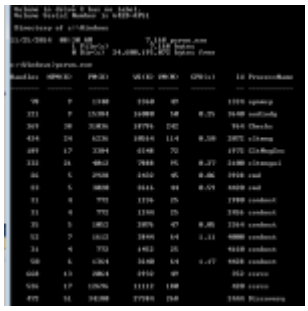
```
//Prep PS for string output and invoke
pipeline.Commands.Add("Out-String");
Collection results = pipeline.Invoke();
runspace.Close();

//Convert records to strings
StringBuilder stringBuilder = new StringBuilder();
foreach (PSObject obj in results)
{
```

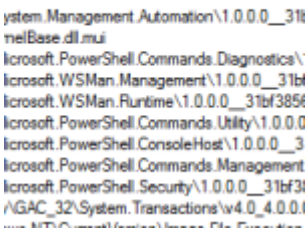
```
    stringBuilder.AppendLine(obj.ToString());
}
Console.Write(stringBuilder.ToString());
```

Compiling this executable and throwing it into the test environment results in an astounding success. Despite all of the preventions on running PowerShell, our executable was able to access the API and run any script we chose. I tested it primarily with Veil-PowerView. 20ish lines of C# demonstrated a pretty rudimentary proof of concept. There are some obvious downsides to this bypass…

- Offensive capability put on disk (although script could be fetched from internet)
- Managed code, namely .NET is easily decompiled
- Whitelist environments defeat this







Defeat The Workaround

This was just too easy… The heavy reliance on .NET had me confident there was an easy way to stop this. In my test environment, I spent some time in Process Explorer and Process Monitor digging around a dump of my malicious application. I confirmed that the custom executable was loading the .NET assemblies to accomplish the PowerShell execution. In the words of Microsoft, “Assemblies are the building blocks of the .NET Framework”. Essentially, assemblies are the libraries that make up the CLI. For the most part, the assemblies are stored in the Global Assembly Cache (GAC) at C:WindowsAssembly. Neat fact, .NET loads the assemblies at run time as they are needed… not upon initial application execution. As I played with the code, I could see the point where the assembly would get loaded. Below are a sampling of the file handles I saw open:

- `C:WindowsassemblyGAC_MSILMicrosoft.PowerShell.Commands.Diagnostics1.0.0.0__31bf3856ad364e35Mic`
- `C:WindowsassemblyGAC_MSILSystem.Management.Automation1.0.0.0__31bf3856ad364e35System.Managemen`

Running quickly back to AppLocker, I added all of the PowerShell specific assemblies from the GAC to the deny policy by hash. To get the assembly DLLs blocked in AppLocker and allow explorer to access the backend assembly libraries, I had to disable the “Cache Viewer” functionality. Definitely be sure to reenable this if you want drag-and-drop install of assemblies. A quick reg command FTW:

```
reg add HKLMSoftwareMicrosoftFusion /v DisableCacheViewer /t REG_DWORD /d 1
```

For some reason, the assembly deny policy still failed to deny execution to my malicious application. At first I thought it might be due to the fact that the DLL was hidden in an assembly, but after testing, that turned out to be false. Other assembly DLLs could be blocked. See the follow up section for more info! The only DLL that I could find that would successfully prevent execution via AppLocker DLL Rules was:

```
C:WindowsMicrosoft.NETassemblyGAC_32System.Transactionsv4.0_4.0.0.0__b77a5c561934e089System.Transact
```

This prevented the execution of my malicious script but the DLL being blocked was the library responsible for all transactional classes (aka the ability to create a resource manager). This was an unreasonable block as tons of legitimate .NET applications require resources. Finally, I moved on to the obvious: rename/move the PowerShell assemblies.

```
mkdir c:windowsassemblydisabled
move c:windowsassemblyGAC_MSILMicrosoft.PowerShell.Commands.Management1.0.0.0__31bf3856ad364e35Micro
```

Obviously this worked and prevented execution of my malicious application. In my mind, this is a limited solution due to the inflexibility of having to install/uninstall assemblies. It would be ideal to have a configurable policy to allow the assembly to be used by certain users or in certain situations but prevent use by the standard user.

FOLLOW UP (added after post)

After posting this originally, several people got involved in discussion via twitter. Before going too much further, I must state that the best policy to use is a whitelist based policy. Blacklisting should be considered nothing more than a stop gap and will be overcome by a dedicated attacker. This post was intended to demo that. With that being said, I rarely even discover companies who have made the jump to any sort of AppLocker policies.

Many thanks for Lee Holmes (@Lee_Holmes) and Carlos Perez (@Carlos_Perez) for the follow up and twitter discussion. Lee Holmes dug in and found out why I was not able to get the PowerShell assemblies working successfully in DLL blacklist mode. He used the AppLocker audit mode combined with Windows Event logs to detect which DLLs were being loaded and blocked based on policies. This will definitely help narrow what specifically you can block. After a bunch of playing with this method, I can vouch it is useful.

To blacklist a .NET assembly, specifically the PowerShell ones, you must block ALL of the related DLLs behind the assemblies. This includes the DLLs in GAC_MSIL and NativeImages if they both are present. To discover the DLL for the PowerShell assembly, you can run this in PowerShell: *[PSObject].Assembly.Location.* There are also several different PowerShell dependencies in the .NET assemblies so feel free to check it out yourself. Using this method, I was able to block the assembly DLL required for the .NET import.

Thats All Folks…

There is some obvious follow on work for this project that I hope to explore:

- Execute/Use the PowerShell API through unmanaged code. There might be a way to access the CLI through a COM object allowing you to integrate PowerShell into your existing RATs and Tools without utilizing the PS executables
- Explore and test different AV/HIPS products with .NET assemblies

Hopefully this stirs some discussion and encourages both the defensive and offensive PowerShell communities to dig deeper. Do not think that blocklisting or setting permissions on a single executable will prevent the use of the PowerShell language. There needs to be an effective and native capability to deny or limit the use of PowerShell by an adversary. As this is proof of concept type of work, please let me know if you have better defensive methods or have seen specific software that works to defend PowerShell really well.
