

≡ MENU

ODDVAR MOE'S BLOG

Notes from My adventures with Windows security

PERSISTENCE USING UNIVERSAL WINDOWS PLATFORM APPS (APPX)

Posted on 6 Sep 2018

TL;DR

Persistence can be achieved with Appx/UWP apps using the debugger options. This technique will not be visible by Autoruns.

Two different approaches exists (registry keys). Listed below are the two techniques for two different apps that starts at logon:

Cortana app:

```
reg add HKCU\Software\Microsoft\Windows\CurrentVersion\PackagedAppXDebug\Microsof
OR
reg add HKCU\Software\Classes\ActivatableClasses\Package\Microsoft.Windows.Cortan
```

People app:

```
reg add HKCU\Software\Microsoft\Windows\CurrentVersion\PackagedAppXDebug\Microsof
OR
reg add HKCU\Software\Classes\ActivatableClasses\Package\Microsoft.People_10.1807
```

When one of the techniques are added the file (cmd.exe) will be executed at logon, since Cortana and People will start at logon. This technique is depending on the correct versions of the apps. Could be clever to add all possible versions of an application.

DESCRIPTION

I used some time to look for ways to evade Autoruns again and I ended up in finding a new method of achieving persistence. New in the terms that I can not find anything about this being abused with the help from Google. If you are reading this and you have some resources on the topic, please share it with me and I will update this blogpost. I decided to send this to MSRC even if I suspected that this would not be serviced. I also sent an email directly to Mark Russinovich. (Awesome guy!)

In this post I will walk you through my process when I discovered this.

This all started while I was looking into the things that start when you logon to Windows and an idea struck my head. What about all these modern apps that starts when the user logs on? I mean, the start menu in Windows is defined as an application. The question I ended up asking myself was, how do the developers debug these applications? After a little Googling I ended up on [this page](#) that described PLMDebug.exe. The interesting part from this page when I looked into this, was the following:

PLMDebug.exe is included in [Debugging Tools for Windows](#).

Copy

```
plmdebug /query [Package]
plmdebug /enableDebug Package [DebuggerCommandLine]
plmdebug /terminate Package
plmdebug /forceterminate Package
plmdebug /cleanterminate Package
plmdebug /suspend Package
plmdebug /resume Package
plmdebug /disableDebug Package
plmdebug /enumerateBgTasks Package
plmdebug /activateBgTaskTaskId
```

Parameters

Package

The full name of a package or the ID of a running process.

DebuggerCommandLine

A command line to open a debugger. The command line must include the full path to the debugger. If the path has blank spaces, it must be enclosed in quotes. The command line can also include arguments. Here are some examples:

```
"C:\Program Files (x86)\Windows Kits\8.0\Debuggers\x64\WinDbg.exe"
```

```
"\"C:\Program Files\Debugging Tools for Windows (x64)\WinDbg.exe\" -server npipe:pipe=test"
```

This really showed promise and I already had the Debugging tools present on my machine, so I started to play with it on a virtual machine I had in Azure.

I decided that I would do my initial testing against the People app, you know... this one:

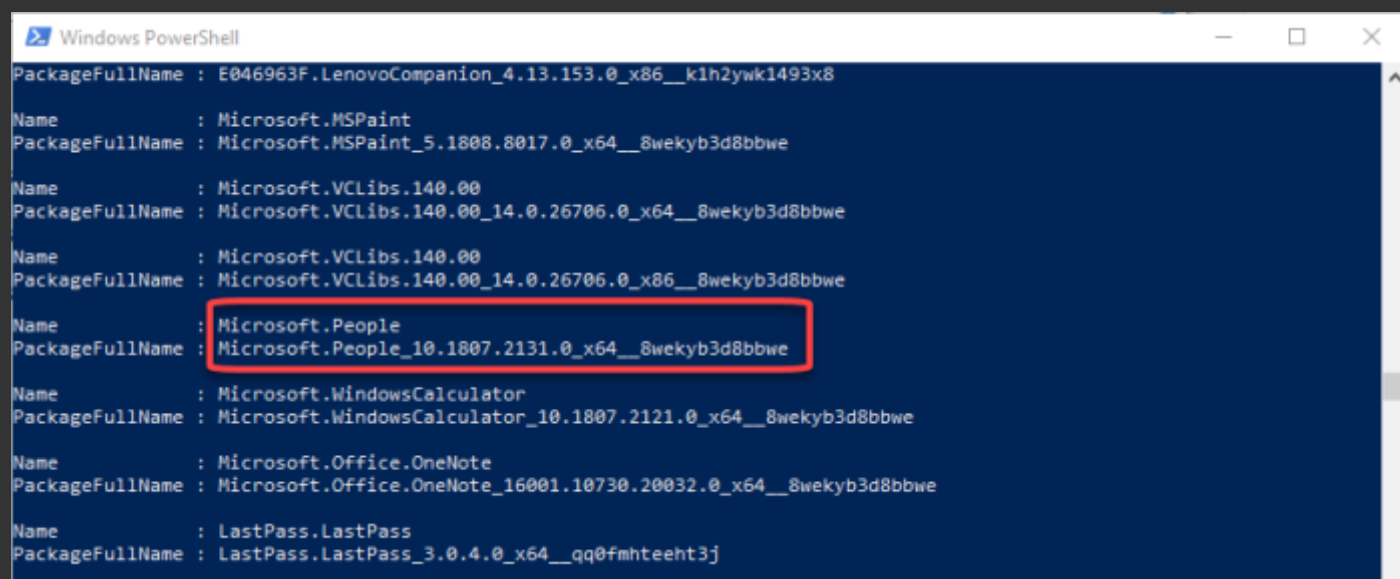


First I needed to figure out the full package name since the PLMDebug command needed it. Since I have previously worked a lot with both SCCM operating system deployment and AppLocker I knew that you can list out all the packages with the Get-AppxPackage cmdlet.

I was not 100% sure what it was named, so I listed all the Appx and only displayed the name and packagefullname using this command:

```
Get-AppxPackage | fl name,packagefullname
```

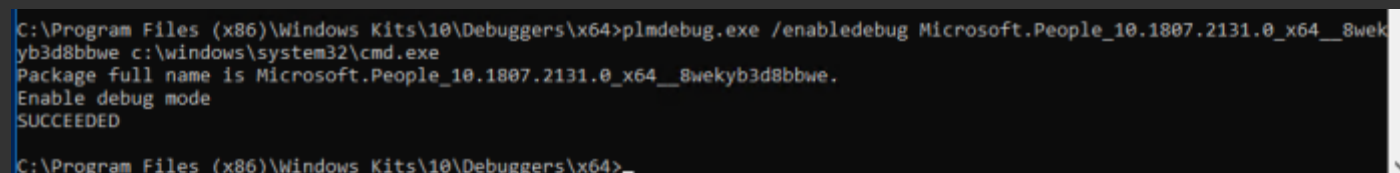
Then I scrolled the output until I found it:



```
Windows PowerShell
PackageFullName : E046963F.LenovoCompanion_4.13.153.0_x86__k1h2ywk1493x8
Name           : Microsoft.MSPaint
PackageFullName : Microsoft.MSPaint_5.1808.8017.0_x64__8wekyb3d8bbwe
Name           : Microsoft.VCLibs.140.00
PackageFullName : Microsoft.VCLibs.140.00_14.0.26706.0_x64__8wekyb3d8bbwe
Name           : Microsoft.VCLibs.140.00
PackageFullName : Microsoft.VCLibs.140.00_14.0.26706.0_x86__8wekyb3d8bbwe
Name           : Microsoft.People
PackageFullName : Microsoft.People_10.1807.2131.0_x64__8wekyb3d8bbwe
Name           : Microsoft.WindowsCalculator
PackageFullName : Microsoft.WindowsCalculator_10.1807.2121.0_x64__8wekyb3d8bbwe
Name           : Microsoft.Office.OneNote
PackageFullName : Microsoft.Office.OneNote_16001.10730.20032.0_x64__8wekyb3d8bbwe
Name           : LastPass.LastPass
PackageFullName : LastPass.LastPass_3.0.4.0_x64__qq0fmhteht3j
```

I then supplied the PackageFullName to the plmdebug.exe using this command:

```
plmdebug.exe /enabledebug Microsoft.People_10.1807.2131.0_x64__8wekyb3d8bbwe c:\w
```



```
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64>plmdebug.exe /enabledebug Microsoft.People_10.1807.2131.0_x64__8wekyb3d8bbwe c:\windows\system32\cmd.exe
Package full name is Microsoft.People_10.1807.2131.0_x64__8wekyb3d8bbwe.
Enable debug mode
SUCCEEDED
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64>
```

I then tried to start the People app from the start menu and voila:

After this was a success I tried to sign out and logon again to see if the cmd window appeared as part of the logon process. This was a bummer, nothing happened. Also when I tried to re-launch the People app nothing happened. Decided that I would do a procmon dive when I ran the PLMDebug command to understand what was going on under the hood. It did not take long until I found out where it was written in the registry.

The registry path from the screenshot is this one (It shows something else in Procmon (HKU\SID), because I am running it as another user):

```
HKEY_CURRENT_USER\Software\Classes\ActivatableClasses\Package\Microsoft.People_10
```

I then jumped into the registry location and exported all the keys. Then I tried to sign out and logon again, re-checked the registry location and it was empty (as expected). Apparently the PLMDebug.exe “flags” something that makes the registry keys go away after logoff. I then decided to try to add the exported registry keys manually and maybe in that way exclude the potential “flags” that PLMDebug.exe sets in addition, that clears the settings at logoff.

After manually importing the registry keys, I signed out and logged on again. When I was logged on again the cmd window popped. (YES!)

I decided to change the persistence to the Cortana application (I did experience some hiccups using the People app early in my testing – but have not experienced it since. Could be my lab setup that had issues), since it is used whenever you logon and everytime you use the search function in the Start menu. I did try with the start menu (Microsoft.Windows.ShellExperienceHost) for a limited time during my research, but I did have some issues with the start menu locking up so I ended up using Cortana as my PoC.

After doing the same “exercise” for Cortana (exporting the registry keys) I decided I would also try to trim down the registry file to a minimum. I did the trimming by removing one registry key at a time until I only had a few left that was needed for the persistence to trigger.

I ended up with these few lines in order to make cmd.exe start at logon using the Cortana app:

```
Windows Registry Editor Version 5.00
```

```
[HKEY_CURRENT_USER\Software\Classes\ActivatableClasses\Package\Microsoft.Windows.
```

```
[HKEY_CURRENT_USER\Software\Classes\ActivatableClasses\Package\Microsoft.Windows.
```

```
[HKEY_CURRENT_USER\Software\Classes\ActivatableClasses\Package\Microsoft.Windows.
```

```
"DebugPath"=hex(2):63,00,3a,00,5c,00,77,00,69,00,6e,00,64,00,6f,00,77,00,73,00,\  
5c,00,73,00,79,00,73,00,74,00,65,00,6d,00,33,00,32,00,5c,00,63,00,6d,00,64,\  
00,2e,00,65,00,78,00,65,00,00,00
```

To make it **easier** to reproduce I created this command:

```
reg add HKCU\Software\Classes\ActivatableClasses\Package\Microsoft.Windows.Cortan
```

After adding this persistence mechanism I decided to check if it was listed in Autoruns and it was not there. I could show you a screenshot, but there is nothing to show. So this is something that **is not detected by Autoruns** as of now. (v13.91)

I did all the testing with a normal user, and I also decided to test this with a user that had local admin rights. I did the same test and the cmd was spawned elevated. This was pretty cool I thought. Decided that I would write a case to MSRC about this new technique, including UAC and the persistence technique.

After a few days I got response from MSRC that the UAC elevation did not work and I had to retry this in my own lab to verify if I had done something different then the information I provided MSRC. It turned out that there are special settings in terms of UAC on the vm's hosted on Azure and that was why it was elevated during logon for an administrator. This did not happen on a "normal" Windows vm hosted on my computer, that was installed from scratch. After this was communicated with MSRC they decided that this did not meet the bar for servicing, since this requires an attacker to already have code execution on the system. (As expected)

If you are planning on running a "custom" executable, it must be able to handle the input the application sends in. If you set the debug application to notepad you can see what it sends into the application:

So, basically -P <number> and -tid <number> will be sent to the "debugger".

All the testing was done with an **up-to-date Windows 10 1803** with the last verification as of September 6, 2018.

FAILED TESTS

Other things I did test during this research that did not give any result:

- Launch file from webdav
- Launch file from HTTP
- Launch file from ADS
- Add registry key to HKLM (admin persistence)

UNCHARTED

There are probably more stuff to find out with Universal Platform Apps.

I have not researched this yet, but this is on my list to look at (feel free to dig into it your self):

- COM hijacking – Seems CLSID is being used
 - HKEY_CLASSES_ROOT\ActivatableClasses\CLSID
 - HKEY_CLASSES_ROOT\ActivatableClasses\Package\Windows.PrintDialog_6.2.0.0_neutral_neutral_cw5n1h2txyewy\ActivatableClassId\Microsoft.Windows.PrintDialog
- Other Apps
 - Did limited testing on the Microsoft.Windows.ShellExperienceHost (Start menu). Had some issues with the start menu hanging.

BONUS DISCOVERY

Also while writing this blogpost and correlating my notes, I discovered that this persistence technique can also be triggered in a much easier way than adding the registry keys mentioned previous in this post. You could simply run the following command to add persistence to the People app:

```
reg add HKCU\Software\Microsoft\Windows\CurrentVersion\PackagedAppXDebug\Microsof
```


Or you could run the following command to add persistence to the Cortana app:

```
reg add HKCU\Software\Microsoft\Windows\CurrentVersion\PackagedAppXDebug\Microsof
```

BLUE TEAM

Hopefully you will detect the attacker long before the persistence is created, but here are a the basic indicators using this technique:

- Software\Microsoft\Windows\CurrentVersion\PackagedAppXDebug
- Software\Classes\ActivatableClasses\Package\<PackageName>\DebugInformation

OUTRO

I do not have any metrics if this technique is already in use in the wild. This post could possibly help to uncover if this is being used or not.

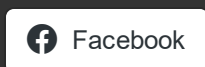
Again, as a reminder I posted this to make it possible to detect it, not to make the world burn. Hopefully this will also inspire further research into UWP/APPX, this could be another untapped source for interesting stuff. I do consider this the modern version of Image File Execution Options.

Hope you enjoyed my post and as always, feedback is appreciated.

DISCLOSURE TIMELINE

- August 17, 2018 – Discovery made –
<https://twitter.com/Oddvarmoe/status/1030465720054960128>
- August 20, 2018 – Report sent to MSRC and acknowledged
- August 23, 2018 – Team reported that the UAC elevation did not work as stated. Asked if anything special was needed to bypass UAC. My attached description did not work.
- August 27, 2018 – I sent that I was not able to reproduce the UAC bypass myself.
- August 28, 2018 – I sent that this only happened on Azure VMs
- August 28, 2018 – Updates forwarded to the team working on the case from my case handler
- September 3, 2018 – I asked if there were any updates to the case
- September 4, 2018 – Case closed, it will not be serviced.
- September 5, 2018 – Mail sent to Mark Russinovich about autoruns
- September 6, 2018 – Mail reply from Mark Russinovich stating it was not a problem for him that I posted these details
- September 6, 2018 – Sent blog draft to MSRC
- September 6, 2018 – All good feedback from MSRC
- September 7, 2018 – Posted

SHARE THIS:



Loading...

PREVIOUS POST

AppLocker for admins – Does it work?

NEXT POST

AppLocker – Making sure that local rules are removed

4 THOUGHTS ON “PERSISTENCE USING UNIVERSAL WINDOWS PLATFORM APPS (APPX)”

Pingback: Researcher finds new malware persistence method leveraging Microsoft UWP apps – Persian Version

Pingback: Researcher finds new malware persistence method leveraging Microsoft UWP apps – My Blog

Pingback: Windows 10 Universal Platform Apps are possible point of persistence for Malware. – Cipher Security MetroState

Morgan Overman says:
11 Sep 2018 at 5:47 pm
This looks promising for a new attack vector. I feel like this needs to be expanded upon.

★ Like
Reply

LEAVE A COMMENT

This site uses Akismet to reduce spam. [Learn how your comment data is processed.](#)



SEARCH

WEBSITE POWERED BY WORDPRESS.COM.