PowerShell Execution Argument Obfuscation (& How It Can Make Detection Easier!)

MARCH 13, 2017

When I started researching PowerShell obfuscation and evasion techniques 1.5 years ago I became frustrated when I found out that many A/V vendors and other detection frameworks and services were writing signatures for specific PowerShell attack frameworks based solely off of PowerShell execution arguments like:

- `-nop -exec bypass -win Hidden -noni -enc`
- `-ep bypass -noni -w hidden -enc`

Fun Fact: In these examples setting -ExecutionPolicy/-ep is redundant since -EncodedCommand/-enc automatically bypasses the execution policy, which is NOT a security boundary!

Some offensive PowerShell frameworks (which won't be explicitly named here) have even kindly left some nice features to write basic signatures off of, like the extra whitespace between *-nop* and *-win* in:

```
-nop  -win hidden -noni -enc
```

However, we as defenders should not become complacent and grow satisfied with only catching attackers that do not modify source code before running the tools they download from Github.

The reason is that good attackers do not remain complacent. If you look at the Github history for Unicorn, an awesome tool from Dave Kennedy (@HackingDave), then you will see that he has changed the syntax of the tool's execution arguments several times as defenders have updated their signatures. He's even left some nice alternate evasion syntaxes in comments for those who actually peruse source code before running tools written by Mr. ReL1K himself:
https://github.com/trustedsec/unicorn/blob/master/unicorn.py#L493-L495

To help break this complacency I focused on this execution argument obfuscation when I built the LAUNCHER component of Invoke-Obfuscation. The primary focus was initially on argument substring and shorthand syntax, randomized case, argument ordering and randomized whitespace between the arguments. So instead of always producing something like

```
-nop -win hidden -noni
```

then it may produce something more like

```
-wINd   hIdDEn   -nOniNT -NOpr
-noProf  -wI  hiDdeN  -noNIntEr
```

These execution argument substrings (like `-NoP, -NoPr, -NoPro, -NoProf, -NoProfi,` and `-NoProfil`) are all valid ways of specifying an execution argument (like `-NoProfile`) because of how PowerShell handles parameter binding. And this parameter binding functionality extends beyond execution arguments. For example, the default Invoke-Mimikatz parameters of `-DumpCreds` and `-DumpCerts` could actually be written instead as `-DumpCr, -DumpCre, -DumpCred` or `-DumpCe, -`

`DumpCer, -DumpCert`. So we as defenders should be diligent to base our detection logic on the lowest common denominator of what we currently know to be syntactically possible.

In October, 2016, I discovered by accident that the -WindowStyle execution argument would accept numerical representation of its flag values: `Normal (0), Hidden (1), Minimized (2)` and `Maximized (3)`. After this realization I updated my detection rules and then added this alternate syntax to the next release of Invoke-Obfuscation (though this technique does not seem to extend to `-ExecutionPolicy` values of `Bypass, Unrestricted,` etc.).

Then last week I read this great blog post called **Pulling Back the Curtains on EncodedCommand PowerShell Attacks** by the Unit42 threat research team (@Unit42_Intel) at Palo Alto Networks. If you haven't read this article article then it is well worth a read, retweet and then one more read. In addition, the Unit42 team published all of the data from their research including the full encoded commands, decoded version, extracted execution arguments, etc. Kudos to the Unit42 team for making this research and data available to the community!

Now when I read this blog post something caught my eye in one of their tables showing a breakdown of the -WindowStyle Hidden syntax across all of the samples in their research (highlighting is mine):

WindowStyle Hidden: (2,083 Samples – 50.8% Coverage)

Used to prevent PowerShell from displaying a window when it executes code. The most used variant "-window hidden" is due to the PowerShell command that the previously mentioned Microsoft Word Documents distributing Cerber are using.

Flag	Count	% of Total
"-window hidden"	1,267	30.90%
"-W Hidden"	315	7.68%
"-w hidden"	159	3.88%
"-windowstyle hidden"	125	3.05%
"-win hidden"	67	1.63%
"-WindowStyle Hidden"	45	1.10%
"-win Hidden"	42	1.02%
"-wind hidden"	40	0.98%
"-WindowStyle hidden"	5	0.12%
"-WindowStyle hiddeN"	5	0.12%
"-windows hidden"	4	0.10%
"-Win Hidden"	3	0.07%
"-win hid"	2	0.05%
"-Window hidden"	2	0.05%
"-Wind Hidden"	1	0.02%
"-Win hidden"	1	0.02%

This table was the first time that I had seen **the argument of an execution argument** using substring obfuscation. After testing and validating that this syntax works, I then updated my detection rules and subsequently updated Invoke-Obfuscation to randomly select any of WindowStyle's arguments' substrings (like the numerical representation, this technique still does not extend to ExecutionPolicy's arguments). So now in Invoke-Obfuscation if you select the WindowStyle Hidden execution argument/value pair then you will get randomized substrings for both WindowStyle as well as Hidden (`H, Hi, Hid, Hidd, Hidde, Hidden, 1`).

BLUE TEAM: HOW POWERSHELL EXECUTION ARGUMENT OBFUSCATION CAN MAKE DETECTION EASIER

Until this past week I focused on keeping track of all known execution argument syntaxes so that I could trigger on particular combinations of execution arguments. However, I realized that there was a simpler (and arguably more effective) detection use case to be made for these obfuscated execution argument substrings: **these execution argument substrings are almost never used legitimately and as such they can be a great indicator of potential obfuscation.**

So for example, `-NoProfile` and its shortest form of `-NoP` are most commonly used in PowerShell commands (for both legitimate and malicious samples). However, just looking for the presence of any of the more obscure, but still valid, substrings in between (`-NoPr, -NoPro, -NoProf, -NoProfi` and `-NoProfil`) would be peculiar enough to warrant closer inspection. This definitely does not mean that we should not continuing monitoring for the more common `-NoP` and `-NoProfile`, but it would seem that we could place more emphasis on these in-between substrings for identifying likely obfuscation attempts.

We can do the same for certain execution arguments' arguments, like WindowStyle's Hidden (`H, Hi, Hid, Hidd, Hidde` and `1`). And in the case of arguments that take arguments (like `WindowStyle` and `ExecutionPolicy`) we can take the path of humility and accept that there are probably smarter and more cunning people that have potentially found more ways to write `Hidden` and `Bypass` (or even `Unrestricted`). With this approach we can look for the absence of all of the syntaxes that we are familiar with. So if we see WindowStyle (with any of its substring syntaxes) and do NOT see any of the normal or obfuscated values that we are currenty aware of then we might want to alert and look more closely at the syntax being used. A rough example of this for WindowStyle could be a case insensitive regex like:

\s+-(W|Wi|Win|Wind|Windo|Window|WindowS|WindowSt|WindowSty|WindowStyl|WindowStyle)\s+[^HMN0123]

If you have experimented with some of the newer LAUNCHER options (Hint: WMIC) from Invoke-Obfuscation then you may see that there are ways to evade the regex above, particularly when you can use characters allowed in a parent process launching powershell.exe to pollute powershell.exe's command line arguments. But the purpose of this sample regex is not to catch everything, but rather to identify a subset of likely obfuscation.

Defense in depth is still the name of the game here. After all, I wouldn't want any defender to take a detection idea (even from me!) and become complacent with it :D Thus the perpetual game of refining, testing and sharing creative detection techniques continues.

Happy hunting!

POWERSHELL, OBFUSCATION, DFIR, EXECUTION ARGUMENTS, DETECTION, BLUE TEAM

Newer Older