Windows Command-Line Obfuscation

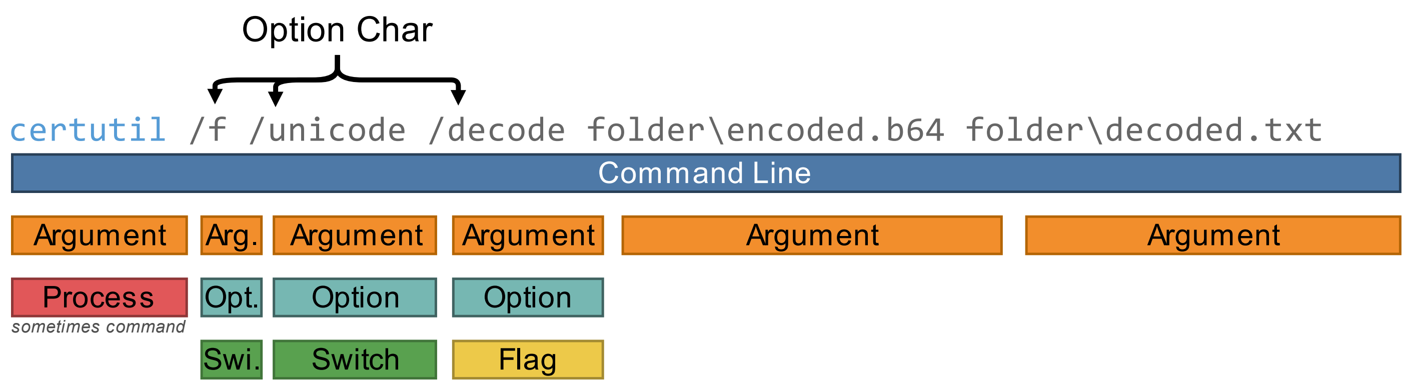
***TL;DR** – Many Windows applications have multiple ways in which the same command line can be expressed, usually for compatibility or ease-of-use reasons. As a result, command-line arguments are implemented inconsistently making detecting specific commands harder due to the number of variations. This post shows how more than 40 often-used, built-in Windows applications are vulnerable to forms of command-line obfuscation, and presents a tool for analysing other executables.*

Command-line arguments

> *He who wishes to be obeyed must know how to command.*
> *Machiavelli (Discourses on Livy III, Chapter XXII)*

It turns out that when it comes to computers, this sixteenth-century quote still very much applies. After all, on most operating systems, processes have a ‘command line’ component, which allows the (initiating) parent process to pass on information to the child process. The command line is accessible to this newly created process, which may change its process flow based on what is found on the command line. This concept is at the core of what makes ‘the computer’: its ability to execute a set of instructions, programs, taking input of sorts.

Despite its fundamental role in computing, there doesn’t appear to be agreement on what to call the various parts that can be found on command lines. Some consider command-line arguments, parameters, options, flags, switches to be one and the same, to some they have different meanings. This post will be using the following terminology:



The line in its entirety is the command line, which is comprised of command-line *arguments* (separated by spaces). Although all are arguments, they have different roles: for example, the first argument is typically the *process* that is being called - or, if you are within a command prompt, this would be the *command*. In the above example it is followed by more arguments. More specifically, the first three are command-line *options*, which typically start with a special character. Of these three options, the first two are *switches* because they don’t require further input, whereas the third is a *flag* because it is followed by further arguments that are ‘inputs’ for this argument.

The *option char* is the character that is used as a prefix on command-line options. On Windows this is typically the forward slash (`/`), on Unix-like systems hyphens (`-`) are predominantly used. This is rather by convention than by rule: because the command line is parsed by the program being executed, developers are completely free in defining in what format arguments should be passed.

Conventions are hard. Computing history has taught us that without standardisation, things tend to end in chaos - and that certainly applies to command lines.

Compatibility

The lack of standardisation in command-line parsing practices has caused confusion amongst users: it is easy to mix up the different practices, leading to unsuccessful executions and much frustration. In an attempt to help users, some programs have been designed to accept multiple conventions. A common example of this is accepting both forward slashes and hyphens as option char, as will be discussed in more detail in the next section.

Another source of confusion is the different character encodings that are available. This has caused all sorts of compatibility issues between old programs (typically only accepting ASCII) and newer programs (nowadays often accepting UTF-8 or Unicode). Some programs attempt to fix these incompatibilities by filtering out certain characters or converting specific characters to ASCII equivalents.

This has led to a situation in which some programs treat a variety of different command-line arguments as one and the same. You could call such instances *synonymous command lines*, since despite the data passed to the process is different, their execution and outcome is identical.

Problems for detection

Although being able to express the same command in different ways may be of help to some users, it tends to make detection and prevention efforts harder. Most threat detection software (AV, EDR, and so on) will monitor process executions, and look for command-line arguments that may be indicative of malicious usage. As attackers will want to go undetected, they may take advantage of the flexibility of the command line in an attempt to evade detection. This may range from small tweaks to bypass keyword-based detection, to full-blown command-line obfuscation to hide the original commands.

A good example of obfuscation on command-line level is the the excellent work of Daniel Bohannon, DOSfuscation [1, 2], which specifically focusses on the Windows command-line prompt CMD. Even though the executing CMD ‘understands’ and executes the obfuscated commands, it may look unintelligble to humans, and monitoring software is also likely to be kept in the dark.

The phenomenon of synonymous command-lines goes beyond DOSfuscation, as it applies to many more programs than just CMD. The key difference here being that you wouldn’t just be fooling the command-line prompt, but the executing program itself. As will be shown later, whilst DOSfuscation efforts may end up in unobfuscated form in the recorders used by detection software, the synonymous command-line arguments will not.

Synonymous command lines: methods

To see this in action, we will now take a closer look at five different methods that can cause synonymous command lines.

(1) Option Char substitution

Consider the Windows executable `ping`. Because the program is a port of the original Unix version, the help page suggests command-line options should use a hyphen as option char, e.g. `ping -n 0 127.0.0.1`. This is inconsistent with most other, Windows-native command-line tools, which use forward slashes. Presumably in an attempt to help confused users, the program also accepts the forward slash as option char: `ping /n 0 127.0.0.1` will work as well.

Most built-in Windows executables that work with hyphens also accept forward slashes, but the reverse is not true. The command `find /i keyword` for example will show all files containing the word ‘keyword’, while `find -i keyword` will result in an error.

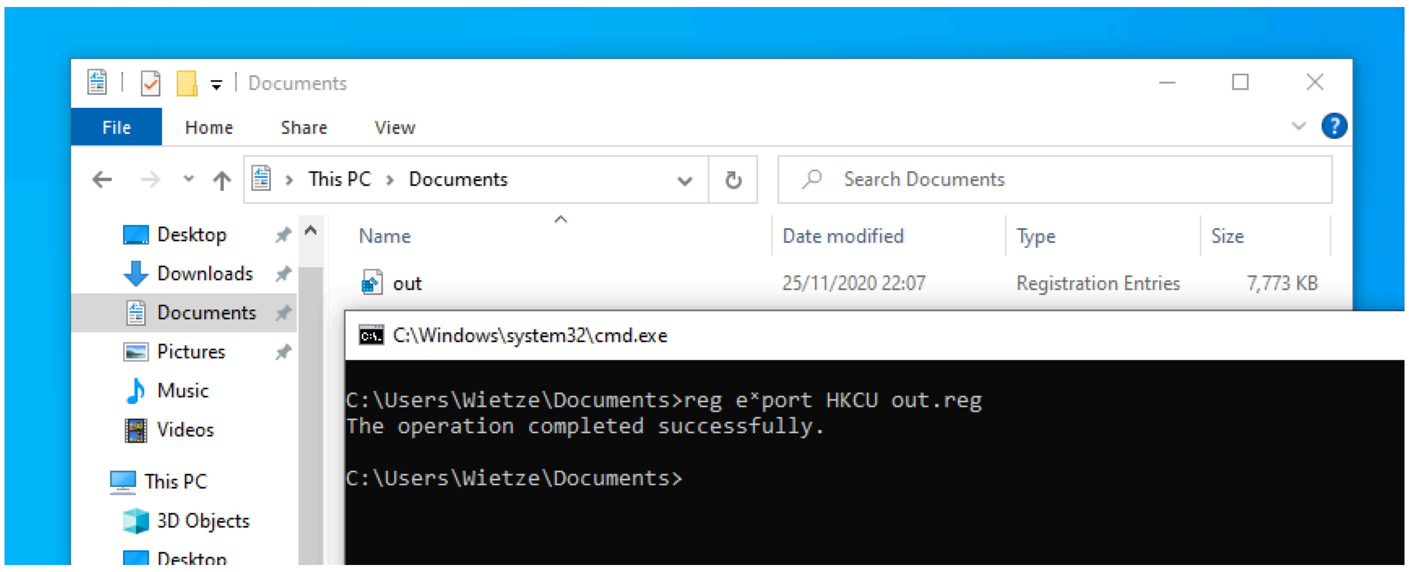
Although forward slashes and hyphens are the most common available options, some programs support even more option chars. `certutil` happens to accept hyphens, slashes, and most Unicode representations of slashes, such as the division slash (0x2215) and fraction slash (0x2044) [3].

As we will see with the other variants, executables rarely document these alternative command-line options, meaning that you will find out about them either by chance, ‘brute force’ or through reverse engineering.

(2) Character substitution

Another approach is to replace other characters (i.e. other than the option char) in the command line with similar characters. Especially when you consider the entire Unicode range, there are many variations of letters also found in the ASCII range that some processes may accept.

Unicode contains a range for *Spacing Modifier Letters* (0x02B0 - 0x02FF) [4], which includes characters such as ˪, ˣ and ˢ. Some command-line parsers recognise these as letters and convert them back to l, x and s respectively. An example of this is `reg`, which treats `reg export HKCU out.reg` and `reg eˣport HKCU out.reg` as equal.



Name	Date modified	Type	Size
out	25/11/2020 22:07	Registration Entries	7,773 KB

C:\Windows\system32\cmd.exe

```
C:\Users\Wietze\Documents>reg eˣport HKCU out.reg
The operation completed successfully.

C:\Users\Wietze\Documents>
```

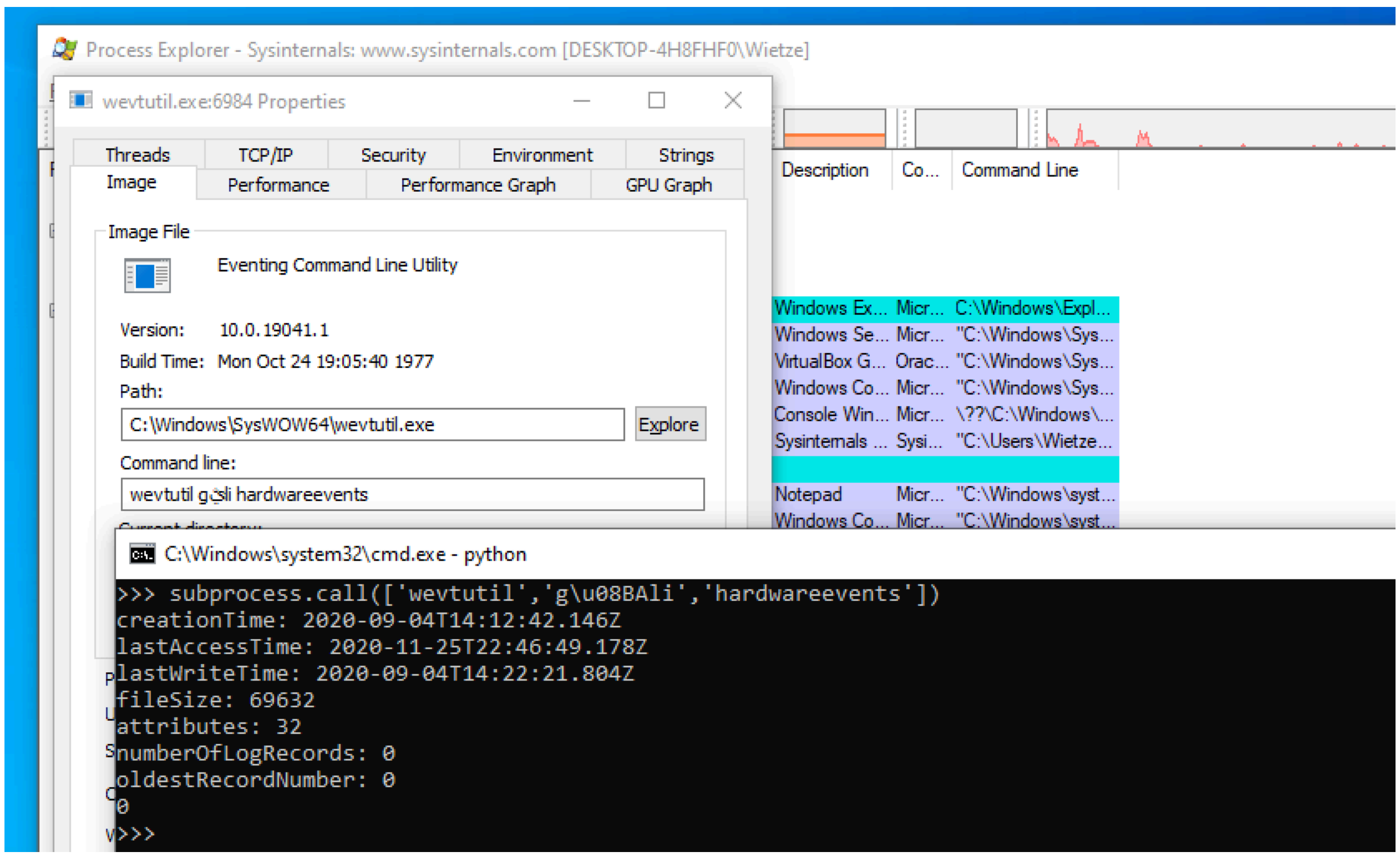
An example of the successful execution of `reg eˣport HKCU out.reg`.

As it turns out, there are more Unicode ranges containing characters that are accepted by some programs.

(3) Character insertion

In a similar vein, it is sometimes possible to insert extra characters into command lines that will be ignored by the executing program. Some executables may remove non-printable characters for example, whilst also certain printable characters may be filtered out.

Windows Event Logs tool `wevtutil` for instance seems to accept command lines that have Unicode characters from certain ranges inserted at random places. So executing `wevtutil gli hardwareevents` and `wevtutil gࢺli hardwareevents` will result in the exact same output, despite the latter containing an Arabic letter in the middle of the first argument.



Process Explorer - Sysinternals: www.sysinternals.com [DESKTOP-4H8FHF0\Wietze]

wevtutil.exe:6984 Properties

Image File: Eventing Command Line Utility

Version: 10.0.19041.1
Build Time: Mon Oct 24 19:05:40 1977
Path: C:\Windows\SysWOW64\wevtutil.exe
Command line: wevtutil gࢺli hardwareevents

C:\Windows\system32\cmd.exe - python

```
>>> subprocess.call(['wevtutil','g\u08BAli','hardwareevents'])
creationTime: 2020-09-04T14:12:42.146Z
lastAccessTime: 2020-11-25T22:46:49.178Z
lastWriteTime: 2020-09-04T14:22:21.804Z
fileSize: 69632
attributes: 32
numberOfLogRecords: 0
oldestRecordNumber: 0
0
>>>
```

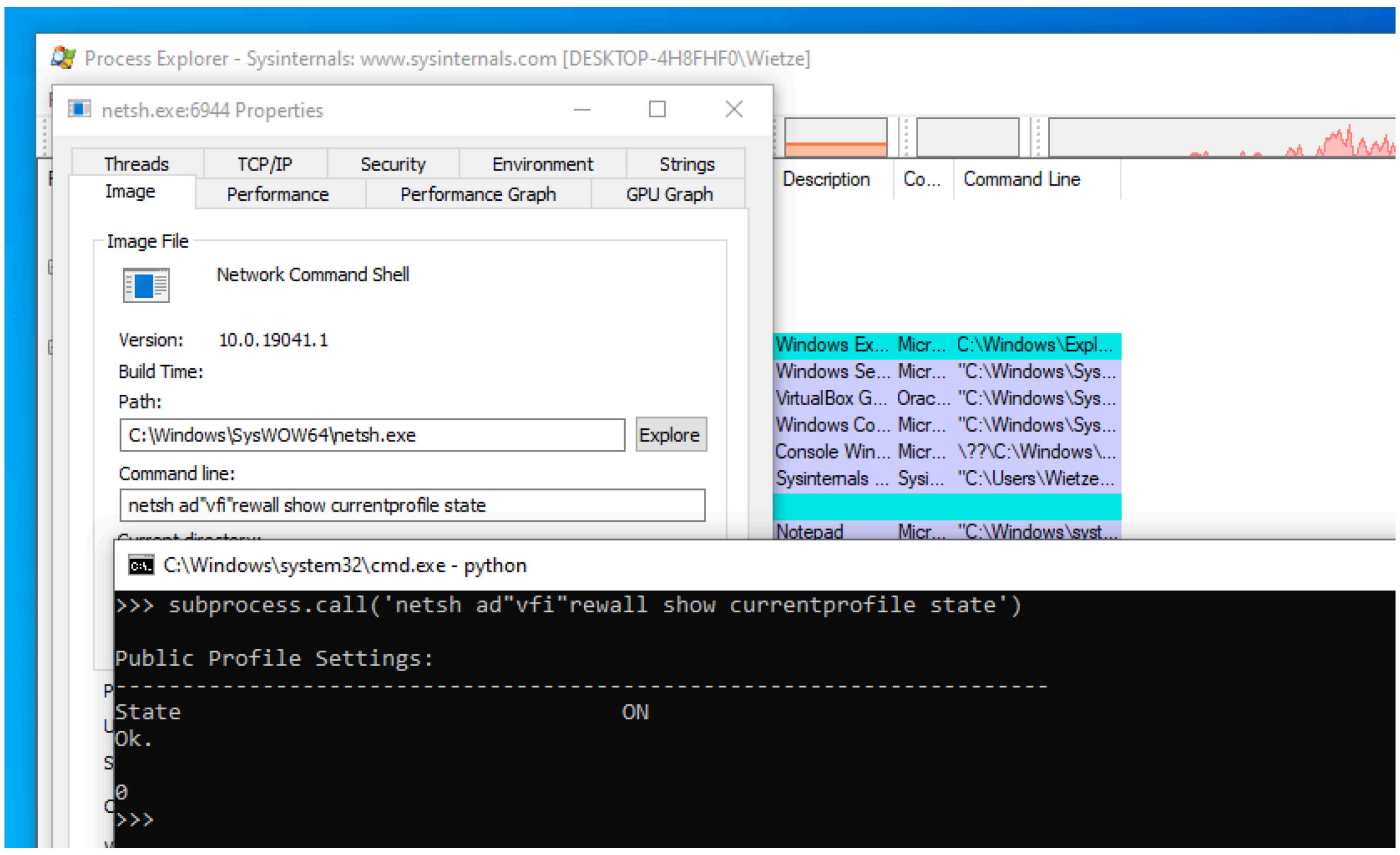
An example of the successful execution of `wevtutil gࢺli hardwareevents`.

Because the characters that can be used for this technique are sometimes unsupported on the stdin of command-line prompts (e.g. because they are non-printable), you may have to insert the character using byte notation. As can be seen in the screenshot, in this case the character is passed correctly to the the process.

(4) Quotes insertion

Another way of manipulating a command line whilst keeping the process flow intact is by inserting quotes. Although this may sound like a subset of the previous technique, a requirement here is that the quotes come in pairs.

You are probably familiar with the concept of putting quotes around arguments. Take `dir "c:\windows\"` for example, which is effectively the same as `dir c:\windows\` due to the lack of spaces. Most programs accept this convention. Less well known is that most programs accept quotes in arbitrary places: the command `dir c:\"win"d""ow"s"` will also work. As long as the number of quotes per argument is even and there are no more than two subsequent quotes, most programs seem to accept this.


Process Explorer - Sysinternals: www.sysinternals.com [DESKTOP-4H8FHF0\Wietze]

netsh.exe:6944 Properties

Image File: Network Command Shell
Version: 10.0.19041.1
Build Time:
Path: C:\Windows\SysWOW64\netsh.exe
Command line: netsh ad"vfi"rewall show currentprofile state

Description	Co...	Command Line
Windows Ex...	Micr...	C:\Windows\Expl...
Windows Se...	Micr...	"C:\Windows\Sys...
VirtualBox G...	Orac...	"C:\Windows\Sys...
Windows Co...	Micr...	"C:\Windows\Sys...
Console Win...	Micr...	\??\C:\Windows\...
Sysinternals ...	Sysi...	"C:\Users\Wietze...
Notepad	Micr...	"C:\Windows\syst...

C:\Windows\system32\cmd.exe - python

```
>>> subprocess.call('netsh ad"vfi"rewall show currentprofile state')

Public Profile Settings:
----------------------------------------------------------------------
State                                 ON
Ok.

0
>>>
```

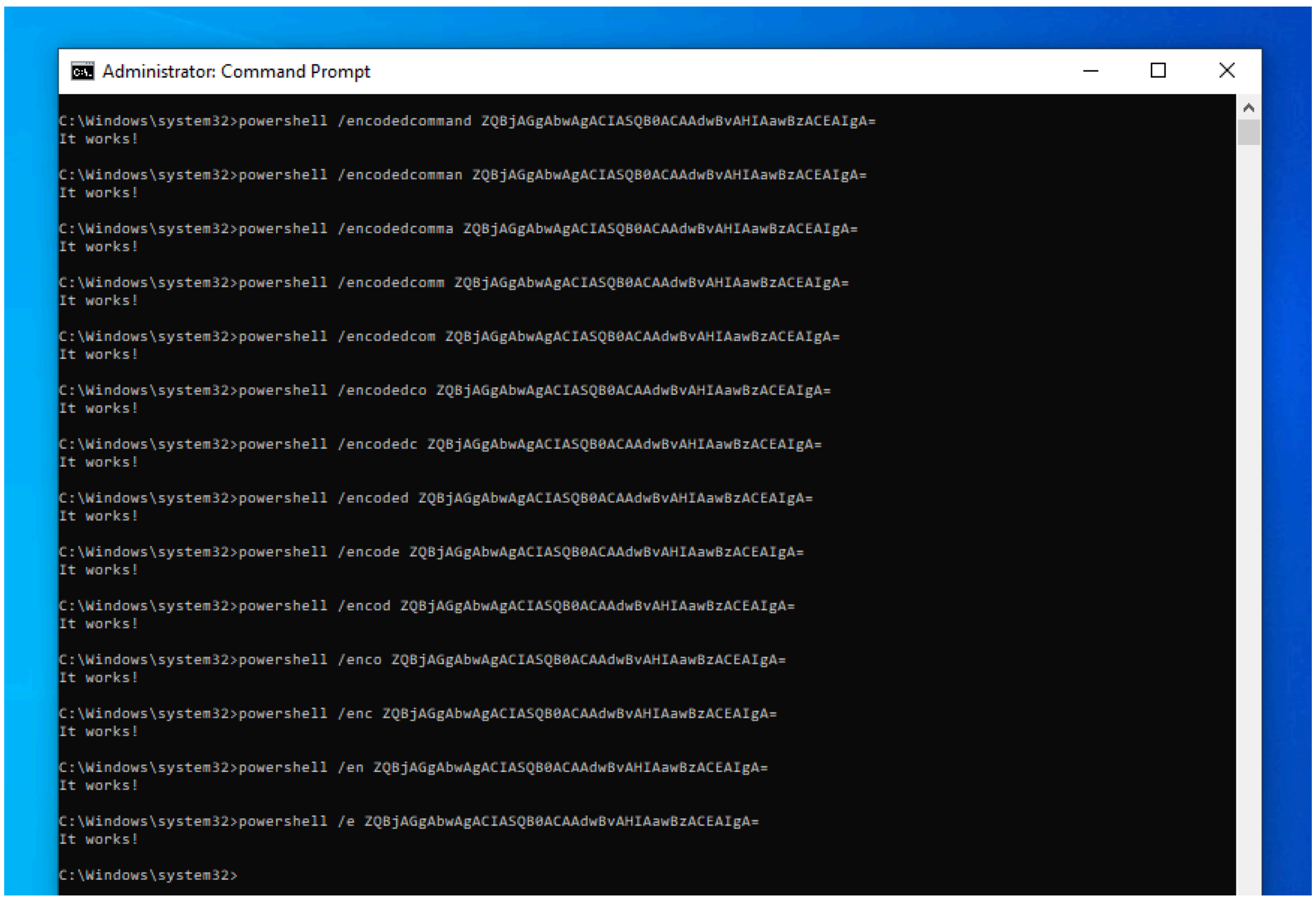
An example of the successful execution of `netsh ad"vfi"rewall show currentprofile state`.

It is worth noting that working with quotes in command prompts can be tricky, as they typically handle quotes themselves before passing it on to the underlying program. A way to work around this in `cmd` for example is to double every quote, so to get the equivalent execution as shown above, you would have to run `netsh ad""vfi""rewall show currentprofile state`.

(5) Shorthands

Having inserted and replaced characters, what remains for us to try is removing characters. Some applications allow 'shorthands' for otherwise lengthy command-line options, making it easier to type them out.

This is a well-known concept in Unix-based tools (e.g. `grep -i keyword` vs `grep --ignore-case keyword`), but on Windows less so. Nevertheless, some programs accept shortened versions. Some programs take a similar approach to Unix and accept a one-letter version (e.g. `cmdkey /l` vs `cmdkey /list`), some accept an otherwise abbreviated version (e.g. `wevtutil gli` vs `wevtutil get-loginfo`), whilst others take a 'wildcard approach'. An example of this is PowerShell, which for many of its keywords allows you to leave off one or more characters at the end of the keyword [5].


Administrator: Command Prompt

```
C:\Windows\system32>powershell /encodedcommand ZQBjAGgAbwAgACIASQB0ACAAdwBvAHIAawBzACEAIgA=
It works!

C:\Windows\system32>powershell /encodedcomman ZQBjAGgAbwAgACIASQB0ACAAdwBvAHIAawBzACEAIgA=
It works!

C:\Windows\system32>powershell /encodedcomma ZQBjAGgAbwAgACIASQB0ACAAdwBvAHIAawBzACEAIgA=
It works!

C:\Windows\system32>powershell /encodedcomm ZQBjAGgAbwAgACIASQB0ACAAdwBvAHIAawBzACEAIgA=
It works!

C:\Windows\system32>powershell /encodedcom ZQBjAGgAbwAgACIASQB0ACAAdwBvAHIAawBzACEAIgA=
It works!

C:\Windows\system32>powershell /encodedco ZQBjAGgAbwAgACIASQB0ACAAdwBvAHIAawBzACEAIgA=
It works!

C:\Windows\system32>powershell /encodedc ZQBjAGgAbwAgACIASQB0ACAAdwBvAHIAawBzACEAIgA=
It works!

C:\Windows\system32>powershell /encoded ZQBjAGgAbwAgACIASQB0ACAAdwBvAHIAawBzACEAIgA=
It works!

C:\Windows\system32>powershell /encode ZQBjAGgAbwAgACIASQB0ACAAdwBvAHIAawBzACEAIgA=
It works!

C:\Windows\system32>powershell /encod ZQBjAGgAbwAgACIASQB0ACAAdwBvAHIAawBzACEAIgA=
It works!

C:\Windows\system32>powershell /enco ZQBjAGgAbwAgACIASQB0ACAAdwBvAHIAawBzACEAIgA=
It works!

C:\Windows\system32>powershell /enc ZQBjAGgAbwAgACIASQB0ACAAdwBvAHIAawBzACEAIgA=
It works!

C:\Windows\system32>powershell /en ZQBjAGgAbwAgACIASQB0ACAAdwBvAHIAawBzACEAIgA=
It works!

C:\Windows\system32>powershell /e ZQBjAGgAbwAgACIASQB0ACAAdwBvAHIAawBzACEAIgA=
It works!

C:\Windows\system32>
```

An example of the successful execution of `powershell /encodedcommand ZQBjAGgAbwAgACIASQB0ACAAdwBvAHIAawBzACEAIgA=` *followed by 13 different shorthands.*

Perhaps apart from the shortest variant `/e`, it seems to me that very few people are helped by accepting these shorthands whilst it does make things more complicated and unpredictable. For example, PowerShell only accepts shortened versions if it doesn't cause ambiguity between another command. For this reason the keyword `/noprofile`'s shortest variant is `/nop`, because `/no` would clash with e.g. `/noexit`. Next to this 'wildcard approach', PowerShell in some cases also accepts acronyms, so despite not shown in the screenshot, `/ec` would also work as a shorthand for `/encodedcommand`.

Finding programs with obfuscatable command lines

As mentioned in the previous sections, there is no standard when it comes to handling command-line arguments. Some programs are very strict, whereas some programs will try very hard to turn whatever it is given into something it can understand. Finding out for a particular program what its command line behaviour is, is problematic too: help pages usually only specify the 'preferred' way of passing arguments (methods 1 and 5) and typically never describe how unexpected characters are treated (methods 2-4).

The most accurate way to determine this seems to be reverse engineering. However, this is a very time-consuming activity, especially when you take the complexity of some of the programs into account. A more pragmatic approach is therefore to simply try all bypass techniques repeatedly with different characters and compare the results.

A proof-of-concept implementation of this approach [6] allows us to quickly analyse which of the five described behaviours apply. A more detailed explanation of the PoC, its underlying assumptions and caveats can be found on the GitHub project page.

For the purposes of this post, the PoC was used to analyse over 40 built-in Windows executables often seen being used in attacks.

Showing **40** entries Search:

Executable ▲	Option Char substitution ▲	Character insertion	Character substitution	Quote insertion	Shorthands
`arp.exe`	✓ (8)	✓ (3)	✓	✓	N/a
`at.exe`	✗	✗	✗	✓	✓
`bitsadmin.exe`	✗	✗	✗	✓	✗
`cacls.exe`	✗	✓ (3,087)	✓	✓	✓
`certutil.exe`	✓ (5)	✓ (6)	✓	✓	✗
`cmdkey.exe`	✓ (1)	✓ (1)	✗	✓	✓
`cmstp.exe`	✗	✓ (3)	✓	✗	✗
`csc.exe`	✓ (1)	✗	✗	✗	✓
`curl.exe`	✓ (3)	✗	✗	✗	N/a
`findstr.exe`	✓ (11)	✗	✗	✓	✓
`fltmc.exe`	N/a	✗	✗	✓	✗
`forfiles.exe`	✗	✗	✓	✓	N/a
`icacls.exe`	✗	✗	✗	✓	✗
`ipconfig.exe`	✓ (1)	✓ (3,370)	✓	✓	✗
`jsc.exe`	✓ (1)	✗	✗	✓	✓

Notes: 1) Windows 10 version 2004 was used for the analysis; 2) For each Windows executable only a single, specific command was used - other commands may result in different outcomes; 3) Entries denoted with an asterisk (*) targeted a command-line argument that accepted *every* variation. In the case of `nslookup -querytype=ALL (...)` for example, the PoC found that the expected output was returned regardless of what character was inserted. It turns out that `nslookup` ignores all letters after the first letter of a command-line option (`-q` in this case), meaning the number of possible permutations is nearly infinite.

Full reports for each tested executable, including the tested commands and the found characters, can be found on GitHub [6].

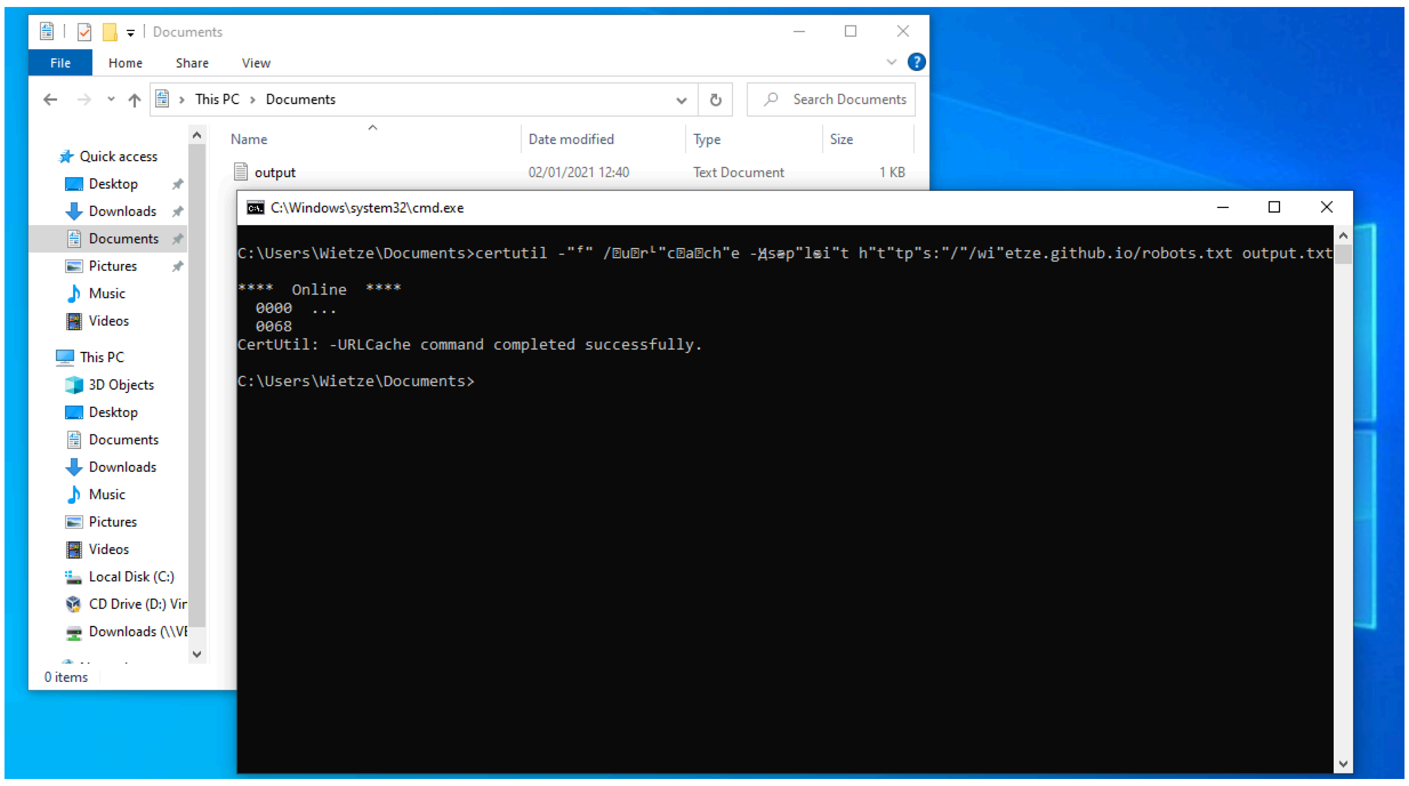
Problems for detection

Knowing all this allows us to take a step back and look at command-line behaviour from a threat-hunting perspective.

If you want to detect executions of programs with specific command-line arguments, synonymous command-line arguments pose a problem. For example, writing a Sigma rule [7] to detect `certutil` executions with the `urlcache` and `f` arguments specified could be expressed as:

```
detection:
    selection:
        Image: "*\\certutil.exe"
        CommandLine|contains|all:
            - "/urlcache"
            - "/f"
    condition: selection
```

The problem is immediately clear: if an attackers uses `-urlcache`, `/urˡcache`, `/urlcache` or `/url"cach"e` - or even worse, a combination: `-u r ˡ"c a ch"e`, the command will still work, but the above rule will not pick up the behaviour. Adding all variants to the rule is not an option, as combining the various bypass techniques will result in thousands if not millions of permutations.



```
C:\Users\Wietze\Documents>certutil -"ᶠ" /u r ˡ"c a ch"e -ʂsep"lɨi"t h"t"tp"s:"/"/wi"etze.github.io/robots.txt output.txt

****  Online  ****
  0000  ...
  0068
CertUtil: -URLCache command completed successfully.

C:\Users\Wietze\Documents>
```

An example of the successful execution of an obfuscated version of `certutil -f -urlcache -split https://wietze.github.io/robots.txt output.txt`.

The first potential solution one may try is to make the rule a bit more generic. For example, solving the option char problem, we could leave the option char out of the definition:

```
detection:
    selection:
        Image: "*\\certutil.exe"
        CommandLine|contains|all:
            - "urlcache"
            - "f"
    condition: selection
```

Whilst this may appear to solve the option char problem, it created a new one. Looking for `urlcache` instead of `/urlcache` is unlikely to cause many false positives, however, looking for just `f` instead of `/f` is likely to generate false positives.

A second potential solution would be to use more clever matching options. Many EDR tools that provide functionality to hunt for process executions tend to be limited to simple 'string contains'-esque features such as expressed in the above examples. Some allow regular expressions, but this still seems to be rare. Although Sigma supports regex values, only 7 of the 30 supported backends appear to actually implement it. Whilst this does not necessarily mean the 23 other platforms do not support it, it at least suggests it may not be an obvious path to go down. A key reason vendors may be hesitant to provide this option in the first place is performance: a normal, compiled regular expression has a complexity of *O(n)*, but through recursion extensions it is possible to create regexes that will never end evaluating. The Sigma project also discourages the use of regexes [8].

For those tools that do support regular expressions, it is possible to work around the option char problem (assuming you replace `[-/\\]` with a list of all option chars):

```
detection:
    selection:
        Image: "*\\certutil.exe"
        CommandLine|re: "^(?=.*[-/\\]urlcache)(?=.*[-/\\]f).+.*"
    condition: selection
```

Regexes may also help efficiently capture shorthand keywords. Unfortunately, character insertion and substitution this is typically not feasible due to the size of options and permutations.

Working around these types of obfuscation is never going to be watertight. More resilient methods are needed.

Robust and resilient detections

To avoid falling into the trap of tunnel visioned detection engineering, there are a few things you can do to achieve robust threat hunting processes.

First of all, make sure rules are defined as 'broadly' as they reasonably can be, catching all possible variations, whilst taking into account the potential increase of false positives. The presented PoC [6] can be used to find variations your rule may allow for.

Secondly, standardising your monitoring tool's output may be beneficial too. If your data is processed in a pipeline before it ends up in your analytics platform, consider adding a separate field that converts all special characters to ASCII equivalents and strips all non-alphanumeric values (to get rid of quotes and option chars). Running your rules over this field might have a higher hit rate.

Furthermore, use data analytics to detect the obfuscation attempt itself. Look at process command lines that contain characters with a low prevalence across your IT estate, compare character density levels, or even simply look for command-line arguments that contain non-ASCII characters and work from there.

Also make sure your focus is broader than process strings alone. Where possible, focus on the actual behaviour that the command line is trying to achieve, such as file creations, registry changes or network connections. For example, rather than trying to capture all possibly malicious `wevtutil` commands, it may be better to look for `wevtutil` writing to unexpected locations (system information discovery? [9]) or checking the event log for event ID 1102 (indicator removal? [10]).

Finally, bear in mind that it won't be possible to detect 100% of the badness 100% of the time - however, the more you you *can* detect, the harder you are making it for an attacker to go completely unnoticed. And as such, taking the above into account will get you one step closer.

"*He who wishes to be obeyed must know how to command*", and he who knows how to command will know how to detect.

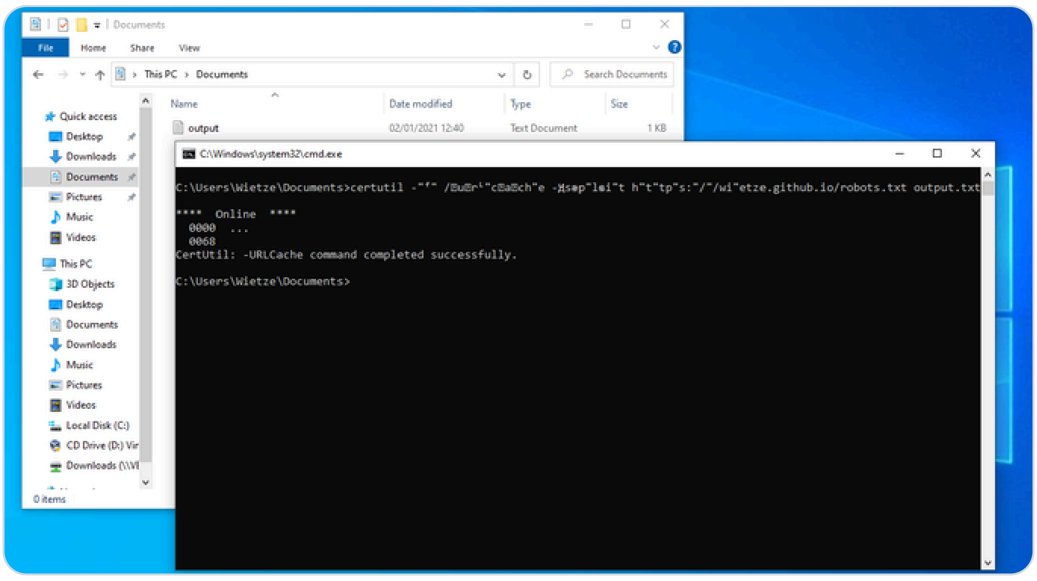
Posted on 2021-07-23

Wietze
@Wietze · **Follow**

After some digging, I found that nearly every common lolbin has command-line features that can be abused to frustrate detection.

How much of your detection content would still work if command-line obfuscation was used? 🔣

Full blog post here👉 wietze.github.io/blog/windows-c...



```
C:\Users\Wietze\Documents>certutil -"f" /ȖuȨr`"cȄaȻch"e -Ӿsəp"lәi"t h"t"tp"s:"/"/wi"etze.github.io/robots.txt output.txt

****  Online  ****
  0000  ...
  0068
CertUtil: -URLCache command completed successfully.

C:\Users\Wietze\Documents>
```

3:23 PM · Jul 23, 2021

485 Reply Copy link

Read 14 replies

Wietze
Cyber troublemaker | Threat Hunting, Detection Engineering & Research | @Wietze

← Go back

Most recent posts

- Why bother with argv[0]?
- Save the Environment (Variable)
- Windows Command-Line Obfuscation
- Hijacking DLLs in Windows
- PowerShell Obfuscation using SecureString