

From a New Year's surprise to a bag of coal - Analysis of mystery PowerShell

By Mo Bustami

Published: 2026-01-05 · Archived: 2026-04-05 16:55:06 UTC

On December 29, 2025 one of my VT hunt rules fired off. I got super excited as that rule was created on the back of a specific PowerShell script from an incident back in the summer and this was the first time it fired off since I created it.

VT Link -

<https://www.virustotal.com/gui/search/8bab6fbed08c3d8d45512b09126dc39bbf02eca8c5a92655baca7ae7dbfb1b4a>

Low detection and still is (3/63) as the writing of this post. I was EXCITED. Is this related to that incident? Is the same actor behind this? Is this an evolution from the one I analyzed in the summer? Well, I got to work.

The script was heavily obfuscated with tons of garbage code as well as tons of integer to character substitutions. But as you scroll down you will be faced with a big byte array that is coded as integers.

```
00000000 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f 50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f 60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f 80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f 90 91 92 93 94 95 96 97 98 99 a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
```

```
00000000 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f 40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f 50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f 60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f 80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f 90 91 92 93 94 95 96 97 98 99 a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
```

Starting to manually analyze the code while tedious, was fruitful as it was a straight integer to character substitution. However, in order to move to the next layer, the byte array had to go through some de-obfuscation.

In specific, it had to be (*Hint: you can use the below in CyberChef to quickly decode it*):

1- Convert it from Decimal, this will produce what looks like a Base64 encoded code. this make sense as the PS script does have fromBase64 decode function.

2- Convert from Base64 but this produce gibberish. hmmm, going through the code you can notice that there is an XOR function.

3- The code needs to be XORed with the hardcoded key of "AMSI_RESULT_NOT_DETECTED" or in hex "414D53495F524553554C545F4E4F545F4445544543544544"

When i found this, i actually got excited, as researching this specific XOR key. it seemed to be associated with different campaigns and malicious actors so I thought I was onto something juicy. Oh, How I was soooo wrong.

The resulting code was another layer of obfuscated PowerShell code but was just a straight up *fromBase64*.

The resulting script was back to the same level of obfuscation of the first script, big chunk of byte array from integer. Buuuut, there was a familiar three letters at the end. Of course in order to get the next stage, I had to replace the *IEX* with *Write-Host*

My trusty [TryItOnline](#) tool was up to the challenge, If you are not aware, this online tool is amazing for emulating and running all kind of code in a 'safe' manner. The output script was interesting as it included some URLs, some steganography and some interesting techniques.

Basically, the script was reaching out to different URLs with PNG extension, these PNGs included some code embedded in the picture and the script was basically concatenating the different parts to create another script. My friendly neighbor LLM describes it better as:

*"This PowerShell script is a highly sophisticated **Steganography Downloader**. Its primary goal is to download a seemingly harmless image file, extract hidden malicious code from the pixels of that image, and execute it in memory.*

Here is a simple breakdown of the steps it takes:

1. Preparation and Evasion

- **Dynamic API Loading:** *Instead of using standard PowerShell commands, it manually "builds" access to Windows system files (`kernel32.dll` and `wininet.dll`). It does this to bypass security software (EDR/Antivirus) that monitors common command patterns.*
- **Defining Custom Delegates:** *It creates "Dynamic Methods" in memory. This is a common technique used by advanced malware to call low-level system functions without leaving a footprint on the hard drive.*

2. Downloading the "Carrier" Image

- *The script contains a list of URLs pointing to image hosting sites (e.g., `pbrd.co` , `iili.io`).*
- *It downloads one of these images into the computer's memory. To a network monitor, this looks like a user simply viewing a picture on a website.*

3. Steganography Extraction (The Secret Sauce)

This is the most complex part of the script. Once the image is downloaded, the script:

- **Reads Pixel Data:** It looks at the individual pixels of the image.
- **Least Significant Bit (LSB) Extraction:** It iterates through the pixels and extracts the very last bit (the 1s and 0s) of the color data.
 - By changing only the last bit of a color, the image looks identical to the human eye, but it can hide thousands of bytes of data.
- **Reassembling the Payload:** It takes these bits and reconstructs them into a binary file.

4. Encryption and Decompression

The data extracted from the image is still not ready to run. The script performs two more steps:

- **XOR Decryption:** It uses a hardcoded 16-character key (`bbL3FzeMEJM07zAN`) to decrypt the data.
- **Gzip Decompression:** It "un-zips" the decrypted data to reveal the final, readable script."

The three URLs the script tries to reach are

`hxxps://gcdnb[.]jpbdr[.]co/images/WHNPY6lHys76.png?o=1'`

`hxxps://iili[.]io/fWW0l4f.png'`

`hxxps://s6[.]jimgcdn[.]dev/Y10TA8.png`

The resulting script which is now layer 5 i think is you guessed it, very obfuscated with the same level of obfuscation (integer/character substitution, very large integer/byte array and XOR'ing). BUT, like two layers before, *IEX* makes an appearance, so replacing that with Write-Host and running it locally as my [TryItOnline](#) could not handle the size of the script produce the next level.



The final stage was, again, you guessed it, another obfuscated script with a larger byte array. But this was the final level. Staging the shellcode loader to run in memory and deploy the payload which according to Gemini is a CobaltStrike loader.

Yes, I was lazy and only did static analysis up until the last layer and the last stage needed a dynamic analysis since the code was performing a

"Technical Breakdown of the Decryption

- **Obfuscation:** The obfuscation style is the same, using variable randomization and string joining to hide API calls³.
- **Decryption:** The decryption routine remains the same custom stream cipher—a rolling XOR where the key updates based on the previous encrypted byte⁴.

2. Shellcode Analysis (New Payload)

While the loader is the same, the **shellcode output** produced by this script is different from the first one. Analysis of the decrypted bytes reveals a different Command & Control (C2) configuration.

Extracted IoCs (Indicators of Compromise)

The following IoCs were extracted from the shellcode payload within this specific file:

Type	Value	Description
C2 Domain/IP	192[.]168[.]45[.]188	The listener IP address for this specific beacon ⁵ .
Port	80	This payload uses HTTP rather than HTTPS ⁶ .
User Agent	Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; InfoPath.2)	A legacy Internet Explorer user agent string ⁷ .
URI Path	/admin/get[.]php	The check-in URI for the beacon ⁸ .
URI Path	/news[.]php	The URI used for data exfiltration/POST requests ⁹ .
Process Spawning	sysprep[.]exe	The shellcode is configured to spawn and inject into <code>sysprep.exe</code> for persistence ¹⁰ .

3. Campaign Association

This sample further confirms the previous attribution:

- **Attribution: Offensive Security (OffSec) Training.**
- **Justification:** The C2 IP `192[.]168[.]45[.]188` belongs to the same private **PEN-200/300 lab subnet** as the previous sample¹¹.
- **Difference:** The first script was likely from a "Web Delivery" or "HTTPS" exercise, while this script is from a "Standard HTTP" or "Legacy Compatibility" exercise, given the use of Port 80 and a Windows XP-era User Agent¹².

Conclusion: This is another training artifact from a penetration testing student. It uses the same sophisticated evasion loader but connects to a different internal lab listener."

For clarity, the reason the above Gemini response is comparing is because I asked it to first analyze a second sample that triggered on January 4. Response is below

"Based on the indicators extracted from the shellcode and the script's behavior, I can attribute this sample with high confidence.

Attribution: Student / Red Team Training (Offensive Security)

This script and shellcode are **not** part of a malicious campaign by a nation-state actor (APT) or a criminal ransomware gang. Instead, they originate from a **Penetration Testing / Red Teaming Lab Environment**.

Evidence 1: The C2 IP Address (`192[.]168[.]45[.]201`)

The IP address `192[.]168[.]45[.]201` is the "Smoking Gun."

- **Context:** `192[.]168[.]45[.]x` is the specific subnet used for the **Offensive Security (OffSec)** labs, particularly for the **OSCP (PEN-200)** and **OSEP (PEN-300)** certification courses/exams.
- **Why it matters:** This IP is a private VPN address reachable only by students connected to the OffSec network. It is not a public internet IP, meaning this "malware" cannot function outside that specific lab environment.

Evidence 2: The Script Techniques (Direct Syscalls)

The PowerShell script uses advanced techniques to bypass Endpoint Detection and Response (EDR) systems:

- **Dynamic Resolution:** It manually finds the address of `ntdll[.]dll` in memory.
- **Direct System Calls:** It calls `NtAllocateVirtualMemory` and `NtCreateThreadEx` directly, bypassing user-mode hooks in `kernel32[.]dll` (a technique known as "Hell's Gate" or generic Syscall invocation).
- **Curriculum:** These techniques are core components of the **OSEP (PEN-300)** Evasion Techniques and Breaching Defenses course, further confirming the OffSec origin.

Evidence 3: The C2 Profile (Cobalt Strike Defaults)

The extracted Indicators of Compromise (IoCs) match the **Default jQuery Malleable C2 Profile** for Cobalt Strike, which is widely available on GitHub and used as a standard template in training:

- **URI:** `/jquery-3[.]3[.]1[.]min[.]js` (Mimics a legitimate JavaScript library request).
- **URI:** `/submit[.]php` (The default POST URI for Cobalt Strike).
- **User-Agent:** The generic Chrome/58 string is the default setting in many cracked or standard versions of Cobalt Strike used in labs.

Summary of Findings

Feature	Analysis	Conclusion
C2 IP	192[.]168[.]45[.]201	Offensive Security Lab VPN (OSCP/OSEP)
Payload	Cobalt Strike Beacon	Standard Red Team Tool
Profile	jQuery / <code>submit.php</code>	Default/Generic Configuration
Loader	Syscalls / Rolling XOR	OSEP/Malware Dev Coursework

Final Verdict: This file is a **Red Team artifact**, likely generated by a student or a penetration tester working within the **Offensive Security (OffSec)** lab environment. It poses no threat to the public internet because its Command & Control server is on a private, inaccessible network."

So here you have; the file my rule triggered on was an OSEP related flag or lab. The rule triggered on another file on January 4, 2026 also with low detection (1/64)

That's it! Happy new year everyone :)

Source: <https://sec0wn.blogspot.com/2026/01/from-new-years-surprise-to-bag-of-coal.html>