See what it's like to have a partner in the fight.

redcanary.com/blog/amsi

January 12, 2022

Try to imagine the following scenarios:

- A process exhibits suspicious behavior but there are no relevant command-line artifacts. How do you make sense of the root cause of the suspicious behavior?
- A PowerShell process downloaded and executed a payload in memory. The command and control (C2) URL is present but there is no execution context beyond that. What exactly was downloaded and executed?
- A **`DotNetToJScript`** payload loaded a .NET assembly in memory. How did the script do it and what did it load?
- A child process spawned from the WMI service `wmiprvse.exe`. How?
- A heavily obfuscated script executed, and it is a challenge to make any sense of what it's actually doing.
- An Office document has a heavily obfuscated macro, and you spend hours trying to untangle how code was loaded.

Adversaries evolve by investing in tradecraft that abuses features that have little-to-no preventative controls or detection optics in place. Take, for example, script and Office macro-based tradecraft. Historically, AV engines and EDR products have engaged in an effective arms race against file-based malware but in-memory payloads have been a challenging blind spot. Some vendors rose to the challenge by injecting code into processes and hooking functions commonly abused by attackers. This strategy was and remains effective, however, it poses an interoperability and maintenance burden subject to instability, concerted evasion by mature adversaries, and disapproval from operating system vendors.

Fortunately, Microsoft recognized the need to improve in-memory optics, while at the same time offering a stable interface for themselves and third party vendors to tap into. What resulted is the Antimalware Scan Interface (AMSI).

AMSI is undoubtedly one of the most significant improvements in endpoint security optics. Whether executable code resides on disk or executes purely in memory no longer makes a difference and as a result, vendors and enterprise defenders now have a vastly wider field of view when it comes to detecting elusive behavior.

What is the Antimalware Scan Interface?

AMSI is an application programming interface (API) developed by Microsoft that enables developers to opt in to sending content to vendor endpoint security agents, regardless of the content's origination, on disk or in memory.

When an application attempts to submit content to be scanned by a vendor agent (referred to as an AMSI provider), the application loads `amsi.dll` and calls its `AmsiInitialize` and `AmsiOpenSession` functions in order to establish an AMSI session. The content to be scanned is then submitted via the `AmsiScanString` or `AmsiScanBuffer` functions.

On the receiving end of the content, a vendor's AMSI provider DLL receives and makes a determination about the content. Vendors can have more than one AMSI provider formally registered in the `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\AMSI\Providers` registry key using their associated COM GUID values. When the provider DLL makes a decision on the content, it passes on its decision to the original application with an `AMSI_RESULT` enum value. If the return value is `AMSI_RESULT_DETECTED` and the underlying scanning engine hasn't already performed a preventative action, it is up to the submitting application to decide what it wants to do with the content that was classified as malicious.

To better understand the structure of AMSI provider registration, let us consider how the Microsoft Defender AV AMSI provider is registered on a stock Windows 10 image:

- Within the `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\AMSI\Providers` key, there is a subkey with a name of `{2781761E-28E0-4109-99FE-B9D127C57AFE}`
- To identify the corresponding AMSI provider DLL associated with that GUID value, reference the following registry value: `HKEY_CLASSES_ROOT\CLSID\{2781761E-28E0-4109-99FE-B9D127C57AFE}\InprocServer32 - (default)`. The content of that registry value is `%ProgramData%\Microsoft\Windows Defender\Platform\4.18.2110.6-0\MpOav.dll`. So, `MpOav.dll` is the vendor-specific (Microsoft being the vendor in this case) AMSI provider DLL that is responsible for receiving and making a decision about content passed to it by applications.

What applications are designed to send content to AMSI?

The following applications automatically opt in to AMSI content scanning if supported:

- **PowerShell:** instrumented in **System.Management.Automation.dll**
- **VBScript:** instrumented in `vbscript.dll`
- **JScript:** instrumented in `jscript.dll`, `jscript9.dll`, and `jscriptlegacy.dll`
- **VBA macros in Office documents:** instrumented in `VBE7.dll`
- **Excel 4.0 macros:** instrumented in `excel.exe` and `excelcnv.exe`
- **Exchange Server 2016:** instrumented in `Microsoft.Exchange.HttpRequestFiltering.dll`
- **WMI:** instrumented in `fastprox.dll`
- **.NET in-memory assembly loads:** instrumented in .NET 4.8+ in `clr.dll` and `coreclr.dll`
- **Volume shadow copy operations:** instrumented in `VSSVC.exe` and `swprv.dll`
- **User Account Control (UAC) elevations:** instrumented in `consent.exe`

Note: UAC buffers are logged to a different ETW provider (Microsoft-Antimalware-UacScan event ID 1201)

How are AMSI events formatted?

AMSI optics provide a great service to defenders looking to build robust detection logic from AMSI events. In order to maximize our detector breadth, it is important to understand what data is available, how it's formatted, and what fields are the most relevant and why. Since AMSI is a dynamic runtime, defenders will want to instrument AMSI activities and then analyze that content to gain behavioral insights. Think of the testing process as much like dynamic analysis. In fact, many modern dynamic analysis platforms now provide AMSI callback data as part of the dynamic execution.

There are two methods of receiving and interrogating AMSI data:

1. **Advanced:** Implement and register an AMSI provider DLL. This is the supported method of receiving and interrogating AMSI data and what anti-malware vendors use.
2. **Intermediate**: Subscribe to the Microsoft-Antimalware-Scan-Interface Event Tracing for Windows (ETW) provider (event ID 1101).

There are trade-offs with either of the AMSI event sources above. Building your own AMSI provider is a high barrier of entry, but, once installed, you'll have persistent and ongoing AMSI buffer collection. Leveraging the ETW traces from the AMSI interface means you'll need to remember to start and stop the AMSI collection. We do not recommend leveraging either of these AMSI event sources on production systems. More on that under the **Mitigating detective controls** section below.

One of the advantages with ETW collection is that you can use built-in tools to collect content. When an AMSI scan event is triggered, the following fields are captured in an AMSI ETW event:

`session`

- **Data type:** Pointer
- **Description:** A pointer value indicating the handle value of the AMSI scan session. These values increment over time. Some AMSI events may have the same session value, indicating that related content executed.

`scanStatus`

Data type: Byte

Description: A byte value that can be set to either 0 or 1. This value is expected to be 1.

`scanResult`

Data type: Unsigned 32-bit Integer

Description: An integer value indicating the result of the content scan. The following values are supported:

- `0 - AMSI_RESULT_CLEAN` : indicates that the content is known good
- `1 - AMSI_RESULT_NOT_DETECTED` : indicates that the content was not detected
- `0x8000 - AMSI_RESULT_DETECTED` :- indicates that the content triggered a detection and is considered malicious
- `0x4000 - AMSI_RESULT_BLOCKED_BY_ADMIN_BEGIN` : indicates that an administrator policy blocked the content
- `0x4FFF - AMSI_RESULT_BLOCKED_BY_ADMIN_END` : indicates that an administrator policy blocked the content. These values can be anywhere between and including 0x4000-0x4FFF

`appname`

Data type: Unicode string

Description: The name of the application that submitted the content to be scanned. This field is important, as it is used to distinguish content types. For example, PowerShell content is presented differently from WMI operation data. The following `appname` values have been identified:

- **PowerShell:** `PowerShell_<POWERSHELLPATH>_<POWERSHELLVERSION> -`
  Example:
  `PowerShell_C:\WINDOWS\System32\WindowsPowerShell\v1.0\powershell.exe_10.0.19041.1` :
  indicates that `System.Management.Automation.dll` submitted PowerShell script code to be scanned
- **VBScript:** indicates that vbscript.dll submitted a sequence of one or more VBScript operations to be scanned
- **JScript:** indicates that `jscript.dll`, `jscript9.dll`, or `jscript legacy.dll` submitted a sequence of one or more JScript operations to be scanned
- **WMI:** indicates that `fastprox.dll` submitted a sequence of one or more WMI operations to be scanned
- **DotNet**: indicates that `clr.dll` submitted a full in-memory .NET assembly PE file to be scanned
- **coreclr:** indicates that `coreclr.dll` submitted a full in-memory .NET assembly PE file to be scanned
- **VSS:** indicates that `VSSVC.exe` or `swprv.dll` submitted a volume shadow copy deletion or resize event to be scanned
- **Excel:** indicates that `excel.exe` submitted Excel4 macro operations to be scanned
- **Excel.exe:** indicates that `excelcnv.exe` submitted Excel4 macro operations to be scanned
- **OFFICE_VBA:** indicates that `vba7.dll` submitted Visual Basic for Applications (VBA) macro operations to the scanned
- **Exchange Server 2016:** indicates that Exchange Server via `Microsoft.Exchange.HttpRequestFiltering.dll` submitted an email header to be scanned. Prior to submitting the email header, Exchange will attempt to first strip personally identifiable information from the email header.

`contentname`

Data type: Unicode string

Description: If the content originated from a file on disk, the `contentname` field is populated with the full path of the disk-based content. If the content originated in memory, this field will be blank.

`contentsize`

Data type: Unsigned 32-bit integer

Description: The size of the content array in bytes. This is the content length that is used to hash the content array.

`originalsize`

Data type: Unsigned 32-bit integer

Description: This field, in practice, is expected to be identical to `contentsize`.

`content`

Data type: Byte array

Description: A byte array of the raw content. Depending upon the application supplying the buffer, the byte array will either be a unicode-encoded string or a binary-formatted data. Currently, the only two content types that are binary-formatted are `DotNet` and `VSS` events. A `DotNet` event consists of the entire portable executable (PE) contents of the in-memory loaded .NET assembly. A `VSS` event contains a currently undocumented structure consisting of a volume shadow copy ID and information about the operation performed: deletion or resizing.

`hash`

Data type: Byte array

Description: The SHA256 hash of the content

`contentFiltered`

Data type: Boolean

Description: As of this writing, this value is hardcoded to false in `amsi.dll`.

Generating and analyzing AMSI events

This section will focus on dynamic analysis of attacker behaviors through the lens of AMSI scan buffer optics. To start capturing AMSI events for dynamic analysis, run the following command from an elevated command prompt:

```
logman start AMSITrace -p Microsoft-Antimalware-Scan-Interface Event1 -o AMSITrace.etl -ets
```

The above command will begin to log any AMSI events to `AMSITrace.etl`. When you are done generating events, stop the trace with the following command:

```
logman stop AMSITrace -ets
```

Upon stopping the trace, you can begin to investigate AMSI events that fired. The resulting `.etl` file can be loaded into Event Viewer or parsed with the PowerShell Get-WinEvent cmdlet. A limitation of these options, however, is that they don’t interpret the event fields. This is why we wrote a helper function Get-AMSIEvent in order to present these events in a more readable fashion.

Generating example AMSI data

The following section will highlight a few instances where different AMSI events are generated and subsequently interpreted. When reviewing this data, be mindful of the behaviors executed and how they map to the AMSI scan optics in the ETW trace. Depending on the AMSI source, you may receive the entire buffer unaltered (PowerShell) or a subset of the behaviors to include (but not be limited to) suspect function calls or assemblies loaded into memory. Understanding these AMSI scan behaviors is a very important defender perspective to have when looking at the best way to detect and prevent threats.

WMI events

The following test commands were executed to perform WMI persistence (T1546.003) and to generate WMI AMSI events:

```
wmic /NAMESPACE:"\\root\subscription" PATH __EventFilter CREATE Name="MimikatzStart",
EventNamespace="root/cimv2", QueryLanguage="WQL", Query="SELECT ProcessName FROM
Win32_ProcessStartTrace WHERE ProcessName=\"mimikatz.exe\""

wmic /NAMESPACE:"\\root\subscription" PATH ActiveScriptEventConsumer CREATE
Name="WriteDateTimeWithMimikatz", ScriptingEngine="VBScript", ScriptText="Set
FSO=CreateObject(\"Scripting.FileSystemObject\"):Set File =
FSO.CreateTextFile(\"C:\Windows\Temp\text.txt\"):File.WriteLine FormatDateTime(now):File.Close"

wmic /NAMESPACE:"\\root\subscription" PATH __FilterToConsumerBinding CREATE
Filter="__EventFilter.Name=\"MimikatzStart\"",
Consumer="ActiveScriptEventConsumer.Name=\"WriteDateTimeWithMimikatz\""
```

Upon running these commands, Get-AMSIEvent presents the following event data (presented in JSON format for easier reading):

```
{
    "ProcessId":  8248,
    "ThreadId":  1320,
    "TimeCreated":  "\/Date(1632412619050)\/",
    "Session":  0,
    "ScanStatus":  1,
    "ScanResult":  "AMSI_RESULT_NOT_DETECTED",
    "AppName":  "WMI",
    "ContentName":  "",
    "ContentSize":  618,
    "OriginalSize":  618,
    "Content":
"ActiveScriptEventConsumer.GetObject();\nActiveScriptEventConsumer.GetObject();\nSetPropValue.Name(\"
 FSO=CreateObject(\"Scripting.FileSystemObject\"):Set File =
FSO.CreateTextFile(\"C:\\Windows\\Temp\\text.txt\"):File.WriteLine
FormatDateTime(now):File.Close\");\n",
    "Hash":  "6B2EDEC830EF3806C37DF36328A91F9F3E65827FEBCFACF19EF02E15576E7914",
    "ContentFiltered":  false
},
{
    "ProcessId":  9180,
    "ThreadId":  5032,
    "TimeCreated":  "\/Date(1632412619234)\/",
    "Session":  0,
    "ScanStatus":  1,
    "ScanResult":  "AMSI_RESULT_NOT_DETECTED",
    "AppName":  "WMI",
    "ContentName":  "",
    "ContentSize":  326,
    "OriginalSize":  326,
    "Content":
"__FilterToConsumerBinding.GetObject();\n__FilterToConsumerBinding.GetObject();\nSetPropValue.Consume

    "Hash":  "A1256C43FC2AABF22E97A6CDD66B17865E005B49270E0DEA62F56DF75EF745BB",
    "ContentFiltered":  false
}
```

Analysis and conclusions

When observing detection optics from the above WMI AMSI callback, take note of a few important items. The `AppName` maps to the expected behavior we executed, WMI. The `ContentName` of the WMI execution is blank. This piece of data can be useful when mapping the fileless threat type, which is indirect file activity. `ContentSize` and `OriginalSize` match and are populated with the length of the data in the `Content` field. The `Content` from the WMI callback contains similar data as the original test that was executed.

What conclusions can we draw from this data? Doing a frequency analysis on how often WMI-sourced AMSI telemetry is happening in your environment is a great place to start. You could also drill down a level deeper and ask how often indirect/fileless WMI events are happening in your environment. Finally, you have optics into WMI event consumer creation and execution of the consumer in a new WMI AMSI callback. Searching for generic strings that we know will be present in WMI event consumer callbacks would be very valuable when looking for this persistence method.

PowerShell and .NET events

The following example highlights a neutered ZLoader payload where AMSI logs both PowerShell and .NET assembly load events (The ZLoader payload was removed and substituted with NOP instructions). A very common technique we see threat actors abusing PowerShell for is reflective loading. At a high level, reflective loading is the act of loading code into a process without writing that loaded code to disk. In the Zloader example, the adversaries embed the encoded payloads they want to inject directly into the PowerShell process. Since PowerShell offers direct access to the .NET subsystem on Windows, we will see exactly how this in-memory behavior looks through the optics we receive from AMSI. Hint… It’s INCREDIBLE!

```
$assembly = @"
        using System;
        using System.Runtime.InteropServices;
        namespace inject {
                public class func {
                        [Flags] public enum AllocationType { Commit = 0x1000, Reserve = 0x2000 }
                        [Flags] public enum MemoryProtection { ExecuteReadWrite = 0x40 }
                        [Flags] public enum Time : uint { Infinite = 0xFFFFFFFF }
                        [DllImport("kernel32.dll")] public static extern IntPtr VirtualAlloc(IntPtr
lpAddress, uint dwSize, uint flAllocationType, uint flProtect);
                        [DllImport("kernel32.dll")] public static extern IntPtr CreateThread(IntPtr
lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint
dwCreationFlags, IntPtr lpThreadId);
                        [DllImport("kernel32.dll")] public static extern int
WaitForSingleObject(IntPtr hHandle, Time dwMilliseconds);
                }
        }
"@

$compiler = New-Object Microsoft.CSharp.CSharpCodeProvider
$params = New-Object System.CodeDom.Compiler.CompilerParameters
$params.ReferencedAssemblies.AddRange(@("System.dll", [PsObject].Assembly.Location))
$params.GenerateInMemory = $True
$result = $compiler.CompileAssemblyFromSource($params, $assembly)

# kJCQww== - 0x90 (NOP), 0x90 (NOP), 0x90 (NOP), 0xC3 (RET)
[Byte[]]$var_code = [System.Convert]::FromBase64String("kJCQww==")

$buffer = [inject.func]::VirtualAlloc(0, $var_code.Length + 1, [inject.func+AllocationType]::Reserve
-bOr [inject.func+AllocationType]::Commit, [inject.func+MemoryProtection]::ExecuteReadWrite)
if ([Bool]!$buffer) {
        $global:result = 3;
        return
}
[System.Runtime.InteropServices.Marshal]::Copy($var_code, 0, $buffer, $var_code.Length)
[IntPtr] $thread = [inject.func]::CreateThread(0, 0, $buffer, 0, 0, 0)
if ([Bool]!$thread) {
        $global:result = 7;
        return
}
$result2 = [inject.func]::WaitForSingleObject($thread, [inject.func+Time]::Infinite)
```

Both the PowerShell script code and the resulting compiled .NET assembly are logged as AMSI events and interpreted with Get-AMSIEvent below (trimmed):

```
{
    "ProcessId":  1248,
    "ThreadId":  5136,
    "TimeCreated":  "\/Date(1632424204003)\/",
    "Session":  8851,
    "ScanStatus":  1,
    "ScanResult":  "AMSI_RESULT_NOT_DETECTED",
    "AppName":
"PowerShell_C:\\Windows\\System32\\WindowsPowerShell\\v1.0\\powershell.exe_10.0.19041.1",
    "ContentName":  "",
    "ContentSize":  1516,
    "OriginalSize":  1516,
    "Content":  "$assembly = @\"\nusing System;\nusing System.Runtime.InteropServices;\nnamespace
inject {\npublic class func {\n[Flags] public enum AllocationType { Commit = 0x1000, Reserve =
0x2000 }\n[Flags] public enum MemoryProtection { ExecuteReadWrite = 0x40 }\n[Flags] public enum Time
: uint { Infinite = 0xFFFFFFFF }\n[DllImport(\"kernel32.dll\")] public static extern IntPtr
VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint
flProtect);\n[DllImport(\"kernel32.dll\")] public static extern IntPtr CreateThread(IntPtr
lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint
dwCreationFlags, IntPtr lpThreadId);\n[DllImport(\"kernel32.dll\")] public static extern int
WaitForSingleObject(IntPtr hHandle, Time dwMilliseconds);\n}\n}\n\"@",
    "Hash":  "5EB54624AE51F01253FF16D2C21F18D7E55E7C0FC5EF702380C71640FAAD25ED",
    "ContentFiltered":  false
},
{
    "ProcessId":  1248,
    "ThreadId":  5136,
    "TimeCreated":  "\/Date(1632424204401)\/",
    "Session":  0,
    "ScanStatus":  1,
    "ScanResult":  "AMSI_RESULT_NOT_DETECTED",
    "AppName":  "DotNet",
    "ContentName":  "",
    "ContentSize":  3584,
    "OriginalSize":  3584,
    "Content":
"4D5A90000300000004000000FFFF0000B8000000000000004000000000000000000000000000000000000000000000000000
```

Worth noting is that in the case of the `DotNet` event, the entire .NET assembly PE contents are captured.

Analysis and conclusions

The above example shows two unique optics that AMSI brings to the table:

- full script block logging for PowerShell via the instrumentation of `System.Management.Automation.dll`
- the full PE contents of the .NET assembly that was reflectively loaded

In the first callback, we can see that AppName maps to `PowerShell.exe` . The `ContentName` was blank, which from our WMI example, we know is a fileless execution of PowerShell. This is very common with PowerShell reflective loaders. `ContentSize` and `OriginalSize` match and are populated with the length of the data in the `Content` field. The `Content` field contains the full script block for the PowerShell loader that took place. Finally, you get similar AMSI callback context from the `DotNet` execution but instead of the `Content` being populated with a unicode-encoded string, it contains the bytecode of the `DotNet` assembly loaded into the target process.

What types of detection questions can we start to ask related to this behavior? Some of the behaviors that you can see immediately are the Win32 API calls from the PowerShell callback. How often would you expect to see APIs like `CreateThread` or `VirtualAlloc` present in PowerShell executions in your environment? If you have the ability to apply pattern-matching tools like YARA to content, the `DotNet` assemblies are a perfect candidate to scan for known heuristics.

Mitigating detective controls

By design, AMSI events were not intended to be generically consumed by an enterprise outside of vendor solutions like NGAV or EDR. If AMSI data is not available to you or if you are interested in supplemental data sources, there are still some great options.

Microsoft-Windows-PowerShell/Operational event log

Data source: Event ID 4104 – Global PowerShell scriptblock logging

Starting with PowerShell version 5 and above, you can enable global PowerShell scriptblock logging, which will log all PowerShell script code regardless of whether it resides on disk or in memory.

Level: Verbose

Relevant field[s]:

- `ScriptBlockText` – The full contents of the PowerShell script code that executed
- `Path` – If the script content originated from a file, this field will show the full path to the PowerShell script

Example event data:

```
Creating Scriptblock text (1 of 1):
Invoke-Expression 'Write-Host Hello, world of ambiguous intent!'

ScriptBlock ID: bc08d157-4987-4f8f-9806-105594ec630d
Path:
```

Analytical notes: You can apply similar detection logic as you would from the above ETW PowerShell trace. You'll have high quality optics from PowerShell executions but will be missing optics from other data sources like JScript, DotNet, and Office macros. Depending on the system you are pulling global scriptblock logs from, you may end up exceeding your logging capacity, so be careful. If logging capacity is an issue for you, take a look at the automatic scriptlblock logging solution below.

Data source: Event ID 4104 – Automatic PowerShell scriptblock logging

This is the same event as above but the difference is that global scriptblock logging doesn't have to be enabled for these events to be logged. This event is triggered when PowerShell scriptblock content containing any of the following suspicious words are present:

```
Add-Type, DllImport, DefineDynamicAssembly, DefineDynamicModule, DefineType, DefineConstructor,
CreateType, DefineLiteral, DefineEnum, DefineField, ILGenerator, Emit, UnverifiableCodeAttribute,
DefinePInvokeMethod, GetTypes, GetAssemblies, Methods, Properties, GetConstructor, GetConstructors,
GetDefaultMembers, GetEvent, GetEvents, GetField, GetFields, GetInterface, GetInterfaceMap,
GetInterfaces, GetMember, GetMembers, GetMethod, GetMethods, GetNestedType, GetNestedTypes,
GetProperties, GetProperty, InvokeMember, MakeArrayType, MakeByRefType, MakeGenericType,
MakePointerType, DeclaringMethod, DeclaringType, ReflectedType, TypeHandle, TypeInitializer,
UnderlyingSystemType, InteropServices, Marshal, AllocHGlobal, PtrToStructure, StructureToPtr,
FreeHGlobal, IntPtr, MemoryStream, DeflateStream, FromBase64String, EncodedCommand, Bypass,
ToBase64String, ExpandString, GetPowerShell, OpenProcess, VirtualAlloc, VirtualFree,
WriteProcessMemory, CreateUserThread, CloseHandle, GetDelegateForFunctionPointer, kernel32,
CreateThread, memcpy, LoadLibrary, GetModuleHandle, GetProcAddress, VirtualProtect, FreeLibrary,
ReadProcessMemory, CreateRemoteThread, AdjustTokenPrivileges, WriteByte, WriteInt32,
OpenThreadToken, PtrToString, ZeroFreeGlobalAllocUnicode, OpenProcessToken, GetTokenInformation,
SetThreadToken, ImpersonateLoggedOnUser, RevertToSelf, GetLogonSessionData, CreateProcessWithToken,
DuplicateTokenEx, OpenWindowStation, OpenDesktop, MiniDumpWriteDump, AddSecurityPackage,
EnumerateSecurityPackages, GetProcessHandle, DangerousGetHandle, CryptoServiceProvider,
Cryptography, RijndaelManaged, SHA1Managed, CryptoStream, CreateEncryptor, CreateDecryptor,
TransformFinalBlock, DeviceIoControl, SetInformationProcess, PasswordDeriveBytes, GetAsyncKeyState,
GetKeyboardState, GetForegroundWindow, BindingFlags, NonPublic, ScriptBlockLogging,
LogPipelineExecutionDetails, ProtectedEventLogging
```

These events are logged at the Warning level versus the Verbose level when global scriptblock logging is enabled. When investigating suspicious PowerShell activity, it can be very helpful to triage these events first, as they are more likely to contain malicious code.

Level: Warning

Relevant field[s]:

- `ScriptBlockText` : The full contents of the PowerShell script code that executed.
- `Path` : If the script content originated from a file, this field will show the full path to the PowerShell script.

Example event data:

```
Creating Scriptblock text (1 of 1):
[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','NonPublic


ScriptBlock ID: 1e583f6c-1ea6-498d-84b5-b43bccace6b3
Path:
```

Analytical notes: These warning level logs may occasionally bring back benign IT-related scripts that execute in your environment. Overall, logging on the above wordlist will result in a robust detection strategy focused on detecting common attacker behaviors.

Data source: Event ID 4103 – Pipeline execution event

Whenever Add-Type is called in PowerShell to compile and load .NET code on the fly, a 4103 event is generated, which supplies the full contents of the .NET source code (C#, VB.Net, F#, etc.). While .NET AMSI events will capture the compiled version of the result of Add-Type being called, this event offers the plaintext source code in its entirety. While adversaries don't always call Add-Type within malicious PowerShell code, they do sometimes, and this event presents the full plaintext source code of their payload.

This event was introduced in PowerShell version 5 so it will not populate if an adversary executes an older version of PowerShell.

Level: Information

Relevant field[s]:

- `Payload` : Contains the full plaintext of the .NET source code supplied to Add-Type
- `HostApplication` : The process command line of the PowerShell process that called Add-Type

Example event data:

```
CommandInvocation(Add-Type): "Add-Type"
ParameterBinding(Add-Type): name="TypeDefinition"; value="public class ProbablyEvil {
    public static void DoSuspiciousStuff() {
        System.Console.WriteLine("Hello, world of ambiguous intent!");
    }
}"


Context:
        Severity = Informational
        Host Name = ConsoleHost
        Host Version = 5.1.19041.1237
        Host ID = 176bbfdd-f662-4e03-98a0-f124b89c6714
        Host Application = powershell.exe -NoLogo
        Engine Version = 5.1.19041.1237
        Runspace ID = 1f14244d-8efa-415d-8188-496aa9c7d08d
        Pipeline ID = 17
        Command Name = Add-Type
        Command Type = Cmdlet
        Script Name =
        Command Path =
        Sequence Number = 19
        User = TEST\TestUser
        Connected User =
        Shell ID = Microsoft.PowerShell


User Data:
```

Analytical notes: Module logging offers insights into a trending attack surface area, the .NET Runtime. This source may result in a larger rate of log telemetry on systems leveraging applications written in .NET (i.e., Exchange).

“Windows PowerShell” event log

Data source: Event ID 800 – Pipeline execution details

This event is effectively identical to Microsoft-Windows-PowerShell/Operational event ID 4103 described above but it applies to PowerShell version 3 and above. It will not populate if an adversary executes PowerShell version 2.

Because the “Windows PowerShell” event log is a classic event log, it has the downside of not logging the process ID of the PowerShell process that generated the event. If available, Microsoft-Windows-PowerShell/Operational event ID 4103 is ideal as it is an Event Tracing for Windows (ETW) event log, which captures process ID.

Level: Information

Relevant field[s]:

- `HostApplication` : The process command line of the PowerShell process that called Add-Type
- `CommandLine` : The PowerShell script code that invoked Add-Type
- `Details` : Contains the full plaintext of the .NET source code supplied to Add-Type

Example event data:

```
Pipeline execution details for command line: Add-Type -TypeDefinition @'
.

Context Information:
        DetailSequence=1
        DetailTotal=1

        SequenceNumber=18

        UserId=TEST\TestUser
        HostName=ConsoleHost
        HostVersion=5.1.19041.1237
        HostId=176bbfdd-f662-4e03-98a0-f124b89c6714
        HostApplication=powershell.exe -NoLogo
        EngineVersion=5.1.19041.1237
        RunspaceId=1f14244d-8efa-415d-8188-496aa9c7d08d
        PipelineId=17
        ScriptName=
        CommandLine=Add-Type -TypeDefinition @'


Details:
CommandInvocation(Add-Type): "Add-Type"
ParameterBinding(Add-Type): name="TypeDefinition"; value="public class ProbablyEvil {
    public static void DoSuspiciousStuff() {
        System.Console.WriteLine("Hello, world of ambiguous intent!");
    }
}"
```

Preventive controls

Application control

While there is no silver bullet to preventing all script-based attacks, the built-in application control solutions in Windows, AppLocker, and Window Defender Application Control (WDAC) offer a strong mitigation against the execution of script code that is not explicitly allowed, in addition to mitigating PowerShell downgrade attacks.

When either AppLocker or WDAC is deployed, the following relevant events are logged to the “Microsoft-Windows-AppLocker/MSI and Script” event log:

- **event ID 8028**: a file-backed script or MSI was audited and allowed to execute
- **event ID 8029:** a file-backed script or MSI was enforced and prevented from executing
- **event ID 8036:** a disallowed COM object was blocked from executing. WDAC offers a strong mitigation against COM-based threats in script code
- **event ID 8038:** If a script or MSI was audited or enforced, this event is generated and contains signature information about the script or MSI

Another great feature of AppLocker and WDAC when they are in enforcement mode is that PowerShell will enforce Constrained Language Mode, which severely limits the capabilities of an adversary attempting to execute malicious PowerShell code. We advise against the outright blocking of PowerShell execution, and while an adversary would still be able to execute some PowerShell code in a post-exploitation scenario, all of their activities would be logged with scriptblock logging as described above.

Windows Defender attack surface reduction rules

Attack surface reduction (ASR) adds a powerful set of behavioral rules to Microsoft Defender AV that can be placed into audit or enforcement mode. The following rules are applicable to script, macro, and email-based threats:

- **Block all Office applications from creating child processes**
- **Block executable content from email client and webmail**
- **Block execution of potentially obfuscated scripts**
- **Block JavaScript or VBScript from launching downloaded executable content**
- **Block Office applications from creating executable content**
- **Block Office applications from injecting code into other processes**
- **Block Office communication application from creating child processes**
- **Block Win32 API calls from Office macros**

ASR-triggered events are populated in the Microsoft-Windows-Windows Defender/Operational event log with event ID 1121 (audit) and 1122 (enforcement).

Block Office macros

It should go without saying that if there is no operational need to execute macros (VBA and Excel 4.0), then they can be disabled accordingly.

Related Articles