AMSI Bypass & Logging Bypasses

In this lab you get to put on your hacker hat and play with bypasses. First, you’ll bypass the AntiMalware Scan Interface and second you will maliciously disable script block and module logging.

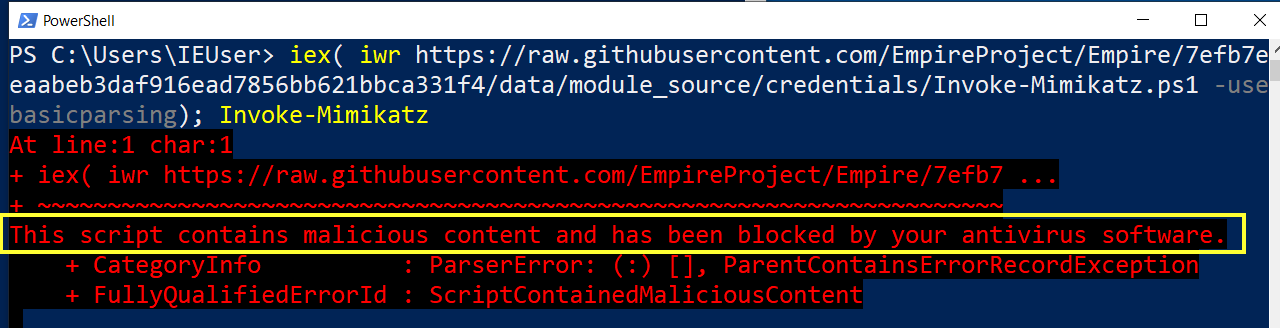
# AMSI Bypass

The AntiMalware Scan Interface (AMSI) is a feature of the Windows Operating System that allows applications to utilize installed Anti-Virus services to analyze code before it is executed. PowerShell is configured to use this interface to verify that code is safe to run before it executes it. In our lab environment, the installed Anti-Virus product is the **Windows Defender** application that comes installed on Windows by default.

In class we introduced 10 popular PowerShell attack tools. Try Running each of the 10 attack tools on your lab VM to see how many of them are blocked because AMSI detects a virus.

Note: Running these commands using a corporate or monitored internet connection could result in the generation of network alerts.

[This file](https://github.com/clr2of8/PowerShellForInfoSec/blob/main/AttackTools/10%20Popular%20PS%20AttackTools.txt) from the class files repository lists the commands you can use to run each attack. Each time you see the “This script contains malicious content and has been blocked by your antivirus software” message you know that AMSI has rejected the code as malicious.



Now that you have seen how many of the attack tools are being blocked by PowerShell’s integration with AMSI, we are going to execute some code to bypass AMSI.

A simple AMSI bypass can be performed in PowerShell by simply telling the current PowerShell session that it’s attempt to “Initialize”, or set-up the connection to AMSI has failed.

Run the AMSI bypass code shown below and repeat the exercise to see what attack tools are allowed to run when we bypass the AMSI interface.

$a = "System.Management.Automation.Amsi"+"Utils"

$b = "amsiInit"+"Failed"

[Ref].Assembly.GetType($a).GetField($b,'NonPublic,Static').SetValue($null,$true)

Text

Description automatically generated

In the image above, we ran the simple AMSI bypass that we learned in class, and this allowed us to run the Keylogger and several of the other scripts, even though they were being blocked by AMSI before.

Interested in some other options for disabling PowerShell’s use of AMSI? There are lots of different ways listed in the GitHub repository [here](https://github.com/S3cur3Th1sSh1t/Amsi-Bypass-Powershell).

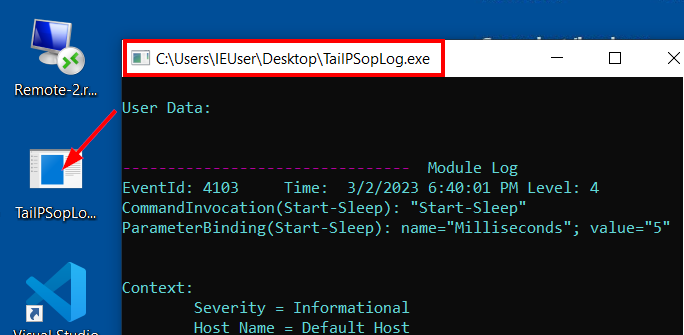
It’s discouraging that AMSI can be so easily bypassed but we have some other security options in PowerShell that we can rely on like Language Modes and “Just Enough Administration” which we learned about in other sections of this class.

# PowerShell Logging Bypasses

We learned that PowerShell logs suspicious script blocks even when script block logging isn’t enabled. We even learned the specific words that PowerShell searches for in the script block to determine whether it is suspicious or not. Here is a command that changes the list of words considered malicious to an empty list (i.e. nothing is considered malicious).

[ScriptBlock]."GetField"('signatures','NonPublic,Static').SetValue($null,(New-Object Collections.Generic.HashSet[string]))

Run the command above and then run the **SuspiciousScript.ps1** from the **PowerShellForInfoSec\**Samples directory. Use the **TailPsOpLog** shortcut on the desktop to watch what gets added to the PowerShell logs in real-time.



Does it get logged as suspicious after we tell PowerShell to not consider any words malicious?

If you watched close, you would have seen that the command we used to disable suspicious script logging was itself, logged as suspicious. This is because **GetField** and **NonPublic** are considered suspicious and we used them before we cleared out the list of suspicious words. What if we just slightly obfuscated the code so that the words don’t exactly match the suspicious words that PowerShell is looking for?

[ScriptBlock]."GetFiel`d"('signatures','N'+'onPublic,Static').SetValue($null,(New-Object Collections.Generic.HashSet[string]))

The changes we made to the key words highlighted in yellow didn’t affect the execution but it did break the string matching and made it so we could avoid having our command logged as suspicious in the PowerShell logs

That is neat, we made it so suspicious script blocks aren’t logged in the script block log, but we still see it logged as part of the module logging. Wouldn’t it be great (as an attacker) if we could easily disable suspicious logging, script block logging and module logging all at the same time.

It turns out that PowerShell uses an Event Tracer object to write to the logs and we learned in class that if we call the **dispose** method on that object, it disables all logging except transcription logging. From your lab VM, Use the **LogMenu** script to Enable all logging and then run the following command in a new PowerShell session.

Text

Description automatically generated with low confidence

([ref].Assembly.definedTypes | ? Name -like "PSEtwLogProvider")."GetFie`ld"('etwProvider','Non'+'Public,Static').getValue($null).dispose()

Now run some commands and scripts from the **PowerShellForInfosSec\Samples** directory, including the suspicious ones. Does anything get logged? You can use the **TailPSopLog** shortcut on the desktop to keep an eye on what is being logged as you execute PowerShell commands.

Text

Description automatically generated

Nothing is logged after we execute our logging bypass to dispose of the ETW object (Event Tracing for Windows).

Again, it is discouraging to see how easily an attacker can disable these valuable logging sources, but it is something we need to be aware of.